

---

# C 编译器剖析

(version 1.03)

邹昌伟 (sheisc@163. com)

---

## 序言

我是在大三的时候开始学习《编译原理》的，授课的是陈意云老师，使用的是老师自己编写的教材。这本教材理论知识非常丰富，每章后面有不少颇有难度的练习题。非常感谢陈老师讲的这门课程，我有限的编译原理理论知识基本上是从这学的。但学完之后，总有一个感觉，即自己掌握的知识比较分散，不能完整的把它串起来。和国内其它高校差不多，这门课的大作业是一个玩具语言的编译器实现，之所以说是玩具语言，是因为语言本身很简单，缺乏很多实用的特性，无法作为实际的编程语言使用。当时就有一个模糊的想法，想自己写一个实用编程语言的编译器，同时这个编译器不能复杂，得适合在一个学期的时间中学习和掌握。但是一直停留在想法这个阶段，也没想好实现哪个语言。

后来的学习和工作中一直打交道的是系统编程领域，使用的基本上是 C 语言。对 C 语言的一些精巧的特性也越来越熟悉，比如说“为什么函数标准的入栈顺序是从右到左”和“`setjmp/longjmp` 是如何实现的”等等。C 语言的使用非常广泛，而且以精炼著称，如果能写一个 C 编译器就很好了。开源的 `gcc` 非常复杂，基本上不适合学习。期间完整的阅读了 `lcc` 的源代码，`lcc` 功能完整，代码精简。但是 `lcc` 代码本身比较难懂，结构不清晰。这个时候就明确下来写一个适合学习的 C 编译器，这个编译器得有如下几个要点：

- (1) 用 C 语言来实现 C 编译器，一个是因为我最熟悉的是 C 语言，另外就是实现自举(`bootstrap`)，这是对编译器的一个很好的测试。
- (2) 代码简洁易懂，结构清晰，适合学生学习和掌握。
- (3) 实现 ANSI C89 标准。
- (4) 一定要开源，因为在学习和工作中受到开源社区的帮助很大。
- (5) 编译器的核心难点和复杂度在后端优化，这是一个简单的用来学习基本原理的编译器，基本不涉及后端优化。

然后从 2007 年开始在闲暇时间构思编写 `ucc`，起的是 `your c compiler` 的缩写，断断续续有一年半的时间完成。完成后很开心，第一时间将其开源在 `sourceforge` 网站，后来因为各种缘故，开源出去后一直也没有继续维护了。

现在非常高兴的看到邹老师以 `ucc` 编译器源码为基础，写了这本理论结合实践的书。这本书不是简单的源码剖析，它以一条编译原理的理论主线把整个知识串起来，辅以实际代码的讲解。我相信这种方式对于在校学生或者编译器爱好者非常有帮助。可能绝大部分程序员在职业生涯中不需要去写一个完整的编译器。但是编译器的一些理论和算法在很多领域得到使用，掌握了编译原理的基础知识之后，会对代码的优化和效率有更好的理解。而且容易掌握一些比较难的语言特性，比如说 `lisp` 语言中的闭包和 `continuation` 这两个概念。

语言是对计算机硬件特性本身的一个抽象，汇编语言抽象的层次低，C 语言抽象的层次高。编译器的整个轴心就是围绕着如何把一个高阶的抽象逐渐变成一个低阶的抽象。这个过程不是一步到位的，是经过了词法分析、语法分析和语义分析等步骤逐渐完成的。每一步可以看做是把一个高阶的抽象语言变成一个稍低阶的抽象语言。带着这样一个主线去学习邹老师的这本书和 `ucc` 源代码，相信大家能很好的掌握。

小米公司 MIUI 首席架构师、UCC 编译器作者  
汪文俊

---

## 前言

从 1999 年在 Turbo C 2.0 下第一次用 C 语言写出 Hello **World** 以来，不知不觉在 IT 相关行当里也混了近 16 年。相信所有上机写过 Hello World 的人当初都会有这样的好奇心：由键盘敲进去的不过是一个个普通的英文字符，C 编译器是如何神奇地发现哪一行出现了什么样的语法错误，更神奇的是编译连接后的可执行程序又是怎么样在操作系统上运行起来的。这些问题也一直萦绕在我脑海里，挥之不去。大学时所学的《操作系统》和《编译原理》课程似乎对这些问题给出了答案，似乎又没有。很遗憾，那时也没有人告诉我，想比较彻底地搞明白这些问题，最好的办法是去读编译器或操作系统的源代码。因为“计算机科学与技术”这一学科，虽然冠上了“科学与技术”，但实际上还是“技术”的成份要来得多些。在所有强调“技术”的工作中，“技术”通常只能来源于长期的动手实践。即便是开车倒库这样的活，也是个技术活，没有长期动手实践是倒不好车的。纸上得来终觉浅，绝知此事要躬行。而不少牛人在大学里上完《操作系统》和《编译原理》课程后就已脱颖而出，比如大牛 Linus 和 Chris Lattner，前者大学时就开启了他的 Linux 操作系统王国，而后者也在研究生期间缔造了如今在业界如日中天的 LLVM 编译器。Apple 公司已经不动声色地把公司的 Object-C 编译器从 GCC 转成了 LLVM 和 Clang，各位的 iPhone 手机中运行的代码可能正是 LLVM 和 Clang 的产物呢。Apple 新推出的 Swift 语言背后站着的仍然是 Charis Lattner 和 LLVM。

能写出工业水准的操作系统和编译器，绝对是大牛。而写出来的操作系统和编译器能被工业界普遍接受，则需要命运和机遇的垂青。大部分的程序员注定是成为高级或低级的“码农”，成为软件生产流水线上的一颗螺丝钉。在生活和工作的压力面前，我们往往习惯于不做那么深入的思考，只要代码能完成需求就可以，至于“为什么”的问题往往有意或无意地，甚至被迫去忽略。就如儿时的梦想，只有在夜深人静时，想着年华已去，看着岁月无情，捧着那泛黄的旧照片时，才发现一直以来自己内心不常去的角落中始终留着那个梦想。

操作系统和编译器就如武侠小说中的《九阴真经》，没看过《九阴真经》的侠客也可以行走江湖，但看过并练成九阴真经的人最终才更有机会登上华山之颠。我们很幸运能生活在互联网时代，不必如上世纪 70、80 年代的欧美程序员那样为了一睹 Unix 操作系统源代码的风采，私下偷偷复印那本后来被称为旷世奇书的《莱昂氏 Unix 源代码分析》。其实，真正旷世之作的是那本书中分析的 Unix 操作系统和用来写 Unix 的 C 语言。当然，我们更不必为了一睹九阴真经，而在江湖中掀起血雨腥风。因为九阴真经就赤裸裸地摆在我们的面前，GCC、LLVM 和 Linux 源代码就毫无保留地躺在那。冲动的我们一定曾经兴奋地下载这些源代码，但面对这动不动就几百万行的源代码时，才发现自己是多么的渺小和无力。其实再复杂的系统，其最初的原型往往是不太复杂的，就如早期的 Unix 操作系统和 C 编译器，而这些原型往往就是最精髓的部分，包括了最初的直觉和创意。与其面对几百万行的源代码感叹自己“too young, too simple”，不如选择一本浓缩的九阴真经。很幸运，只要我们打开 GOOGLE 搜索，我们能找到这些原型版的操作系统和编译器。麻雀虽小，五脏俱全。

如果我们要分析的是开源的相对完整的 C 编译器源代码，而不是一个教学用的 toy，并且希望源代码的行数在一万行左右，那 GOOGLE 给我们的答案中就有这么两个，一个是 LCC 编译器，一个是 UCC 编译器。如果我们还希望代码结构清晰，且函数名等能望文生义，能直接与 C 语言的文法吻合，不必去猜测这个函数名是哪几个单词的缩写，那 UCC 会更胜一筹。但很遗憾，UCC 的原作者汪文俊（wenjunw123@gmail.com）没有去写本关于 UCC 的书，文俊兄其实是最合适的人选。于是，UCC 就如一颗被遗落在角落的珍珠，渐渐地蒙上了尘埃。一万行左右的代码量，UCC 就能实现一个基本完整的 C 编译器，虽然后端只面向 32 位的 X86 平台。UCC 没有如 LCC 那样地可变目标，既可生成 MIPS，也可生成 SPARC 和 X86 代码。虽然偶有 BUG，纵然碧玉有瑕，即便不够知名，但只要把 UCC 这只麻雀解

---

剖清楚，我们已能搞清 C 编译器是如何工作的，我们已能站在编译器实现的角度来品味那经 40 余年而愈久愈香的 C 语言。C 和 C++ 语言的相关书籍中，最经典的如《The C Programming Language》、《C++ Primer》和《The C++ programming language》，之所以经典，很大原因是这些作者本身就是 C 或 C++ 编译器的最早期实现者。

越俎代庖去写本关于 UCC 编译器的书，希望这本书能帮有兴趣读它的人基本搞明白 C 编译器，更好地用好 C 语言这个简洁有力的利器。当然，正如一位在编译一线战斗的朋友所言，“UCC 和 LCC 还停留在编译前端和后端的初级阶段”，编译器的优化是现在工业界更加关注的焦点，UCC 和 LCC 在后端优化上都做得不够。不过，站在广大 C 程序员的角度来看，C 编译器的前端才是 C 编译器与程序员的接口，理解了 C 编译器前端的实现，就能更深刻地理解 C 的语法和语义；而 C 编译器后端，其实是 C 编译器与 CPU 的接口，后端的优化往往为的是在保持语义的前提下，生成速度更快的机器代码。我不是什么牛人，只希望这本书能帮有心人开启一扇看得见、够得着的编译大门，能让 C 程序员能站在编译器实现的角度来看看自己朝夕相伴的 C。全国每年有十万以上的计算机相关专业学生毕业，《编译原理》在大多数毕业生中留下的印象就是很难，很理论化。本书的目的是想把《编译原理》具体化，所谓“伤其十指，不如断其一指”。如果你熟悉 C 语言，并且有基本的数据结构的知识，那就让我们一起开启 C 编译器剖析这趟旅程，编译原理和汇编语言相关的知识我们边走边聊，期盼这趟行程结束后，不论你的下一站去何方，你都能说不虚此行。众里寻她千百度，蓦然回首，She is C。那我们就启程吧。

衷心感谢文俊兄在百忙中替本书写的序言，从 UCC 编译器的源代码中，我受益颇多。在 UCC 源代码的阅读、分析和修改过程中所感受的快乐，像个无形的指挥棒驱使着我把其中的心得和体会写在博客 <http://blog.csdn.net/sheisc> 上。非常感谢清华大学出版社提供的出版机会，十分感谢本书的责任编辑龙启铭老师在本书出版过程中提供的帮助和指导。作为一名在教育一线工作的普通教师，我诚挚地希望拙作能传递这份快乐和收获，能为想深入理解 UCC 编译器的朋友们提供一点点的帮助。书中不足之处，敬请批评指正。

邹昌伟 (sheisc@163.com)

---

## 目录

第 1 章 基础知识.....	1
1.1 语言、文法与递归.....	1
1.2 一个较复杂的文法.....	4
1.3 由文法到分析器.....	6
1.3.1 表达式.....	6
1.3.2 声明.....	12
1.3.3 语句.....	17
1.4 UCC 编译器预览.....	23
1.4.1 UCC 的使用.....	23
1.4.2 UCC 驱动.....	26
1.5 结合 C 语言来学汇编.....	28
1.5.1 汇编语言简介.....	28
1.5.2 整数运算.....	34
1.5.3 浮点数的算术运算.....	39
1.5.4 浮点数之间的比较操作.....	41
1.5.5 指针、数组和结构体.....	44
1.6 C 语言的变量名、数组名和函数名.....	45
1.7 C 语言的变参函数.....	47
1.8 本章习题.....	53
第 2 章 UCC 编译器的基本模块.....	54
2.1 从 Makefile 起.....	54
2.2 词法分析.....	57
2.3 UCC 编译器的内存管理.....	60
2.4 C 语言的类型系统.....	65
2.5 UCC 编译器的符号表管理.....	73
2.6 本章习题.....	81
第 3 章 语法分析.....	83
3.1 C 语言的表达式.....	83
3.1.1 条件表达式和二元表达式.....	83
3.1.2 一元表达式、后缀表达式和基本表达式.....	91
3.2 C 语言的语句.....	99
3.3 C 语言的外部声明.....	106
3.3.1 声明和函数定义.....	106
3.3.2 与声明有关的几个非终结符.....	114
3.3.3 声明说明符和声明符.....	118
3.4 本章习题.....	132
第 4 章 语义检查.....	133
4.1 语义检查简介.....	133
4.2 表达式的语义检查.....	134
4.2.1 表达式的语义检查简介.....	134
4.2.2 数组索引的语义检查.....	137
4.2.3 基本表达式的语义检查.....	142

---

4.2.4 函数调用的语义检查.....	146
4.2.5 成员选择运算符的语义检查.....	157
4.2.6 相容类型.....	159
4.2.7 一元表达式的语义检查.....	166
4.2.8 二元表达式、赋值表达式和条件表达式的语义检查.....	172
4.3 语句的语义检查.....	179
4.4 声明的语义检查.....	183
4.4.1 类型结构的构建.....	183
4.4.2 结构体的类型结构.....	194
4.4.3 结构体和数组的初始化.....	202
4.4.4 内部连接和外部连接.....	211
4.4.5 外部声明的语义检查.....	213
4.5 本章习题.....	217
第 5 章 中间代码生成及优化.....	218
5.1 中间代码生成简介.....	218
5.2 表达式的翻译.....	223
5.2.1 布尔表达式的翻译.....	224
5.2.2 公共子表达式.....	231
5.2.3 通过“偏移”访问数组元素和结构体成员.....	237
5.2.4 后缀表达式的翻译.....	241
5.2.5 赋值表达式的翻译.....	245
5.2.6 一元表达式及其他表达式的翻译.....	250
5.3 语句的翻译.....	252
5.3.1 If 语句和复合语句的翻译.....	252
5.3.2. Switch 语句的翻译.....	256
5.4 UCC 编译器的优化.....	264
5.4.1 删除无用的临时变量和优化跳转目标.....	264
5.4.2 基本块的合并.....	267
5.5 本章习题.....	270
第 6 章 汇编代码生成.....	272
6.1 汇编代码生成简介.....	272
6.2 寄存器的管理.....	278
6.3 中间代码的翻译.....	283
6.3.1 由中间代码产生汇编指令的主要流程.....	283
6.3.2 为算术运算产生汇编代码.....	290
6.3.3 为跳转指令产生汇编代码.....	293
6.3.4 为函数调用与返回产生汇编代码.....	297
6.3.5 为类型转换产生汇编代码.....	302
6.3.6 为取地址产生汇编指令.....	305
6.4 本章习题.....	308
参考文献.....	309
图的清单.....	310
表的清单.....	317
尾声.....	318



# 第1章 基础知识

## 1.1 语言、文法与递归

我们对语言这个概念再熟悉不过了，汉语和英语等都是自然语言。而 C 和 C++ 语言是编程语言，若用更专业的术语，则这些编程语言可被称为“形式语言（Formal Language）”。Formal 这个单词的原意是“正式的”，非常准确地表达了形式语言和自然语言的区别。C 和 C++ 等编程语言之所以可被称为形式语言，其原因在于这些语言的产生过程是先有正式的语法规则，之后才由这些语法规则来产生语言。而汉语和英语则正好相反，原始人从树上刚走下来时，不可能先开会讨论一下语法规则的，最原始的呼喊经数万年的演化和交融，自然而然地形成了汉语和英语等自然语言。若干年之后，广大语文老师们才聚在一起研究语法规则的问题。这种先上车后补票的行为，当然不够 formal 了。这也是为什么自然语言的识别，会成为人工智能领域中一个比较麻烦的问题，而 C 语言的识别从语言一诞生就能得到很好解决的原因之一。

计算机相关学科的人会在大二时学一门叫《离散数学》的课程，里面一个很重要的概念就是集合，这个曾引起第三次数学危机的名词，是个放之四海皆适用的名词。我们要讨论的语言，实际上也是个集合。一般而言，有这么两个方法来表示集合：一个是列举法，一个是描述法。对于个数有限的集合，可以用列举法一一列出；而对于无穷的集合，则多用描述法来表达。例如， $S = \{张三、李四、王五\}$  构成了一个有限集合；而要表达偶数这样的无穷集合，我们则使用  $E = \{x | x = 2n, n \text{ 是非负整数}\}$  这样的描述法。

既然我们要把 C 语言看成一个集合，那么每个合法的 C 源代码就是这个集合里的一个元素，合法的 C 源代码有无穷多个，那我们要用什么样的描述方法来表达 C 语言这个无穷集合？让我们不妨先看看下面这个比 C 语言简单得多的语言。

$$T = \{a, a+a, a+a+a, \dots\}$$

我们不难发现，上述集合 T 是由若干个 a 相加构成，这是个无穷集合，省略号 “……” 的意思是“这个集合的元素太多了，我们列举不下，您自个儿研究已列出的元素，然后去找规律吧”。这种“你猜你猜你猜猜”的描述方法，显然不能让追求完美的我们感到满意。那我们就来探索一下有没有更好的表达方法。让我们先把集合 T 一分为二。

$$\begin{aligned} T &= \{a, a+a, a+a+a, \dots\} \\ &= \{a\} \cup \{a+a, a+a+a, \dots\} \end{aligned} \quad (1-1)$$

我们仔细观察其中的 {a+a, a+a+a, …}，可以发现这个式子跟 T 有点相似。如果把 {a+a, a+a+a, …} 里的每个元素的前缀“a+”移到大括号之前，我们就有

$$\begin{aligned} T &= \{a\} \cup a+ \{a, a+a, a+a+a, \dots\} \\ &= \{a\} \cup a+T \end{aligned}$$

这是一个对集合 T 的递归定义。到此，我们把语言、集合和递归关联到了一起。稍微整理一下，我们把这个无穷集合 T 写为：

$$T \rightarrow a \mid a+T \quad (1-2)$$

式子(1-2)告诉我们，T 由 a 或者 a+T 构成。该式实际上由两个式子构成， $T \rightarrow a$  和  $T \rightarrow a+T$ 。也可分成多行来写，记为：

T:

a  
a + T

如果我们想到事物往往是有阴就有阳，有左就有右，有前就有后，我们就会尝试看看能

不能作其他类似的变换。仔细观察(1-1)式的右部，我们还可以把“+a”作为后缀提取到大括号之后，于是又有了：

$$\begin{aligned} T &= \{a\} \cup \{a, a+a, a+a+a, \dots\} + a \\ &= \{a\} \cup T + a \end{aligned}$$

稍作整理，可以得到下式：

$$T \rightarrow a \mid T + a \quad (1-3)$$

式子(1-2)和(1-3)的区别在于，前者的递归出现在  $a+T$  右侧，而后的递归定义出现在  $T+a$  左侧，所以(1-2)式被称为右递归，(1-3)式被称为左递归。由这两个式子都能产生语言  $T$ ，即都可以生成以下集合中的各个元素：

$$\{a, a+a, a+a+a, \dots\}$$

形如(1-2)和(1-3)这样的式子就被称为产生式。有限的符号却精准地描述了无穷的集合，这非常符合老子在《道德经》所言的“一生二，二生三，三生万物”。让我们享受一下由产生式产生句子的过程。例如要判断字符串  $a+a$  是否是语言  $T$  中的一个合法的句子，即判断字符串  $a+a$  是否是集合  $T$  的一个元素，我们可以从  $T$  出发，作以下推导。下式(1-4)和(1-5)分别是以(1-2)和(1-3)作为产生式进行推导的结果。

$$\begin{aligned} T &\rightarrow a + T \\ &\rightarrow a + a \end{aligned} \quad (1-4)$$

$$\begin{aligned} T &\rightarrow T + a \\ &\rightarrow a + a \end{aligned} \quad (1-5)$$

每一步的推导都是用某个产生式的右侧来替换左侧。下图 1.1 就用一棵树来形象地表述了式子 (1-4) 的推导过程，该树被称为分析树 (parsing tree)。我们可通过这棵树来分析字符串  $a+a$  是否为语言  $T$  的合法句子。对这棵树作进一步观察，可以发现，树中的有些结点可以向外生长，例如其中标为  $T$  的结点；而有些结点则不具备生长能力，例如其中标为  $a$  和  $+$  的结点。因此，我们称式(1-2)中的  $T$  为非终结符，而  $a$  和  $+$  为终结符。式(1-2)由两个产生式构成，即  $T \rightarrow a$  和  $T \rightarrow a+T$ ，它们共同刻画了语言  $T$  的特征，它们是关于如何生成语言  $T$  的法则，所以称之为文法 (grammar)。当然，严格地说，文法由“终结符、非终结符、产生式和开始符号”构成。开始符号是我们要描述的集合对应的非终结符，此处即为  $T$ 。由图 1.1 可以形象地看到，开始符号为树根  $T$ ，这是我们做推导起步的地方，也是语言中所有合法句子的始祖，所以称之为开始符号。实际上如果约定第一个产生式左侧的非终结符为文法的开始符号，则产生式就包含了文法的所有信息。以后的表述中，在不混淆的情况下，我们不去区分文法和产生式。

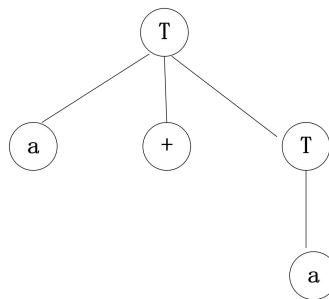


图 1.1 分析树

如果任给一个字符串，我们当然不希望每次都手工来判断它是否为合法的句子。我们需要根据文法来写一个程序来帮我们做这些事情，这个程序被称为语法分析器 (syntax parser)，通常简称为分析器 (parser)。图 1.2 分别给出了为左递归和右递归文法编写语法分析函数的算法，其中的语法图作为一种中间过渡，由语法图我们可很直观地写出相应的分析函数，其基本的原则很简单：

(1) 对于文法中的非终结符, 由于其代表的是一个集合, 我们要判断某个字符串是否属于这个集合, 则会构造一个与非终结符同名的函数来处理。在语法图中遇到非终结符时(对应语法图中的矩形), 例如图 1.2 中的 T, 则直接调用函数  $T()$  来进行语法分析。

(2) 对于文法中的终结符 (对应于语法图中的圆形), 例如图 1.2 中的 a 和 +, 我们则直接调用函数  $Expect$  进行终结符的匹配。

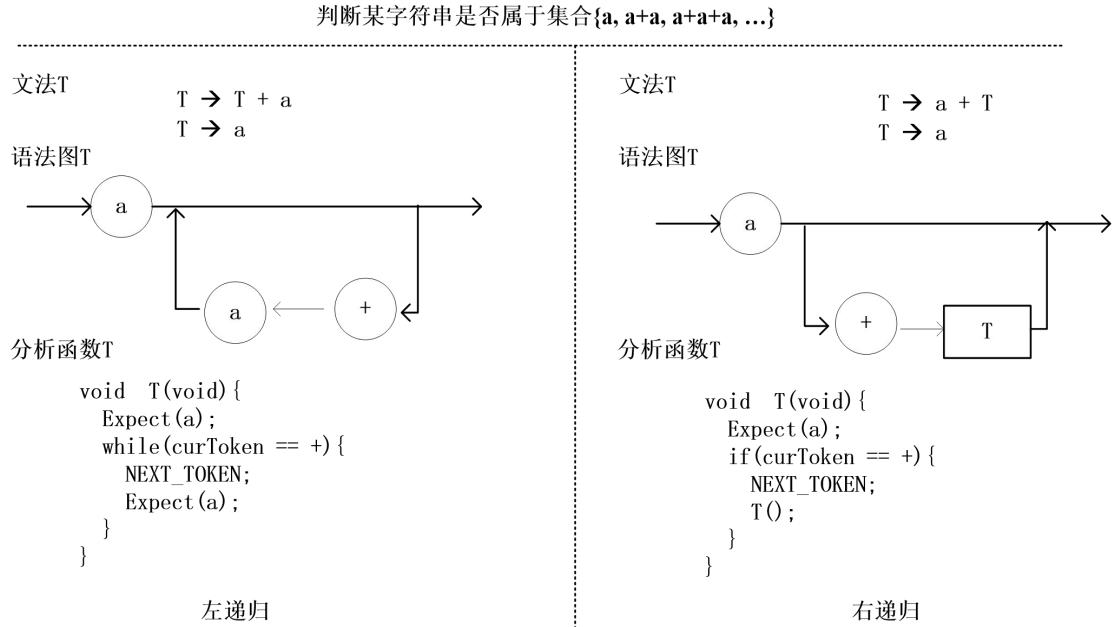


图 1.2 为非终结符 T 编写语法分析函数

图 1.2 中用于终结符匹配的函数  $Expect()$  的算法如下所示,  $NEXT\_TOKEN$  表示读取一个单词并保存到全局变量  $curToken$  中,  $TokenKind$  表示单词的类别。

```
void Expect(TokenKind tk) {
    if(curToken == tk) {
        NEXT_TOKEN;
    } else {
        Error();
    }
}
```

需要注意的是, 若加法运算是左结合的, 对于  $a+a+a$  来说, 按照图 1.2 左侧的分析函数, 我们实际上用到的分析树如图 1.3 右侧所示, 而图 1.3 左侧的分析树是在理论上做推导时用到的, 真正上机实现时我们需要消除左递归。

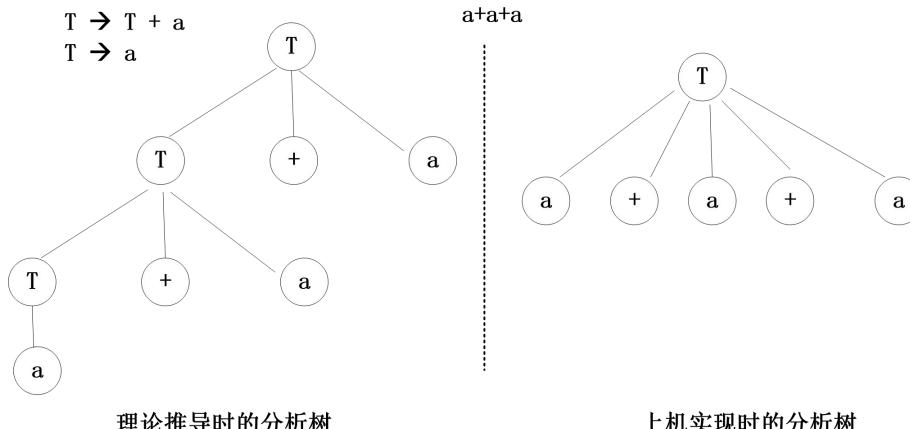


图 1.3 a+a+a 对应的分析树

在熟悉图 1.2 这样的套路后，我们可不必去画语法图，而是直接由文法来编写相应的语法分析函数。在下一小节，我们会引入一个稍为复杂一点的文法，然后我们将介绍如何基于这个文法来写一个语法分析器。

## 1.2 一个较复杂的文法

在上一节中，我们讨论了语言、集合、文法和递归这几者之间的关系，在这一小节中，我们先给出一个较复杂的文法，该文法描述了一个包含“语句、表达式和声明”的程序，如图 1.4 所示。

```

Program → CompoundStatement
Statement → IfStatement | WhileStatement | CompoundStatement
           | ExpressionStatement
IfStatement → if ( expression ) Statement
IfStatement → if ( expression ) Statement else Statement
WhileStatement → while( expression ) Statement
CompoundStatement → { StatementListopt }
StatementList → Statement | StatementList Statement
ExpressionStatement → id = Expression ;
ExpressionStatement → Declaration ;
Expression → AdditiveExpression
AdditiveExpression → MultiplicativeExpression
AdditiveExpression → AdditiveExpression + MultiplicativeExpression
AdditiveExpression → AdditiveExpression - MultiplicativeExpression
MultiplicativeExpression → PrimaryExpression
MultiplicativeExpression → MultiplicativeExpression * PrimaryExpression
MultiplicativeExpression → MultiplicativeExpression / PrimaryExpression
PrimaryExpression → id | num | ( Expression )
Declaration → int Declarator
Declarator → * Declarator | PostfixDeclarator
PostfixDeclarator → DirectDeclarator | PostfixDeclarator [ num ]
                   | PostfixDeclarator ( void )
DirectDeclarator → id | ( Declarator )

```

图 1.4 Program 文法

按照前面的约定，图 1.4 中的第一个产生式左侧的非终结符 Program 为文法的开始符号。让我们来看个例子，就能初步明白这个看起来似乎很复杂的文法究竟刻画了一个什么样的语言。

```
{
    int (*f(void))[4];
    int (*(*fp2)(void))( void );
    if(c)
        a = f;
    else{
        b = k;
    }
    while(c){
        while(d){
            if(e){
                d = d - 1;
            }
        }
    }
}
```

```

    c = c - 1;
}
}

```

请于 <http://download.csdn.net/detail/sheisc/8669715> 下载 ucc162.3.tar.gz, 解压后的目录 ucc\examples\sc 中包含的代码, 即为根据图 1.4 构造出来的语法分析程序。在这个看似复杂的文法中, 不少产生式与第 1.1 节的产生式(1-2)及产生式(1-3)本质上是相同的。

$T \rightarrow a \mid a+T$	(1-2)
$T \rightarrow a \mid T+a$	(1-3)

例如对以下几个从图 1.4 取出的产生式来说, 如果把 AdditiveExpression 视为 T, 把 MultiplicativeExpression 当作 a, 其形式上就和产生式(1-3)是一样的。如前文所述, 产生式(1-3)是一个左递归的产生式。虽然由(1-2)和(1-3)生成的语言是相同的, 但是左递归的产生式隐含了其运算符是左结合的, 而右递归的产生式隐含了其运算符是右结合的。按习惯, 加减乘除这四则运算符都是左结合的。所以我们选取形如(1-3)的产生式来表达加法运算。与 T 所表达的集合类似, AdditiveExpression 由若干个 MultiplicativeExpression 相加构成。换言之, 要进行加法运算, 需要先得到 MultiplicativeExpression。同理, MultiplicativeExpression 由若干个 PrimaryExpression 相乘构成, 这意味着要进行乘法运算, 就要先得到 PrimaryExpression。而 PrimaryExpression 则由标志符 id、数 num 和“加括号的表达式”构成。按这样的层次结构, 以下产生式实际上隐含了加减乘除运算符之间的优先级。加减视为同一优先级, 按结合性进行从左到右的结合; 乘除视为同一优先级, 也采用左结合。用与此类似的方法, 我们还可以引入更多的运算符, 如&和|等。

```

AdditiveExpression → MultiplicativeExpression
AdditiveExpression → AdditiveExpression + MultiplicativeExpression
AdditiveExpression → AdditiveExpression - MultiplicativeExpression
MultiplicativeExpression → PrimaryExpression
MultiplicativeExpression → MultiplicativeExpression * PrimaryExpression
MultiplicativeExpression → MultiplicativeExpression / PrimaryExpression
PrimaryExpression → id | num | (Expression)

```

让我们再来分析图 1.4 中与“语句”有关的产生式。由 Statement 的产生式, 我们可知, 语句由 if 语句、while 语句、复合语句及表达式语句构成。而 if 语句以 if 开头, 后面紧跟左括号, 然后是表达式 expression, 接着是右括号, 之后又是一个语句 Statement。这里, 我们再次在产生式中看到递归定义。

```

Statement → IfStatement | WhileStatement
           | CompoundStatement | ExpressionStatement
IfStatement → if ( expression ) Statement
IfStatement → if ( expression ) Statement else Statement
WhileStatement → while ( expression ) Statement
CompoundStatement → { StatementListopt }
StatementList → Statement | StatementList Statement

```

而 StatementList 对应的产生式实际上形如前面的产生式(1-3)。复合语句

CompoundStatement 中的 StatementList<sub>opt</sub>, 表示 StatementList 是 Optional 的, 即可有可无。接下来, 让我们再来讨论与声明 Declaration 相关的产生式, 在 C 语言中, 我们通过“声明”表达了“C 程序员自定义的某个标志符是什么类型”的概念。

```

Declaration → int Declarator
Declarator → * Declarator | PostfixDeclarator
PostfixDeclarator → DirectDeclarator | PostfixDeclarator [num]
                   | PostfixDeclarator (void)
DirectDeclarator → id | (Declarator)

```

而 PostfixDeclarator 则再次与产生式 (1-3) 神似。其产生式实际上告诉我们

PostfixDeclarator 由 “DirectDeclarator 后面跟上任意多个[num]或者(void)” 构成。而 Declarator 对应的产生式则由“若干个\*后面再跟一个 PostfixDeclarator”组成。通过形如[num]的后缀，我们声明了一个大小为 num 的数组；而通过形如(void)的后缀我们声明了一个无参的函数；而在声明中使用\*，实际上是声明了一个指针。

例如，对于字符串 int aa[30][50]，我们可以按前文画分析树的方法，由 Declaration 出发作推导，最终生成 int aa[30][50]，其中 30,50 皆为数 num，而 aa 为标志符 id。我们还可以由 Declaration 生成 int (\*cc)[3][5]。按 C 的语法，aa 是数组，而 cc 是指向数组的指针。仔细观察，我们发现上述文法中并没有关于数 num 是如何构成的产生式。数 num 和标志 id 实际相当于句子中的单词，我们习惯上把 num 和 id 的识别列入“词法分析”范畴。通常，分析器指的就是语法分析器，在分析器（parser）眼中，只有一个单词。至于如何由基本的英文字母和阿拉伯数字组成单词，我们则在词法分析中进行处理。词法分析器常被称为扫描器（scanner）。

至此，我们慢慢进入了我们的主题“C 编译器剖析”。图 1.4 实际上是一个微缩版的 C 语言文法，其中有常见的语句、表达式和声明。当然，为了简单，我们只引入了一个基本类型 int，个别细节与标准 C 语言的文法也不尽相同。但管中窥豹，已可见一斑。如果理解了上述文法，由文法来构造分析器则是件照图 1.2 “依样画葫芦”的事情。

## 1.3 由文法到分析器

### 1.3.1 表达式

在这一节中，我们来讨论如何为图 1.4 中的文法构造分析器。学英语时，我们从基本的 26 个英文字母入手，然后由英文字母组成各种各样的单词，之后按照语法规则组成句子，之后是段落，然后是文章。同理，我们要写一个语法分析器来识别图 1.4 文法对应的语言时，也需要一个字母表，其中包括我们要定义的语言的基本字符，例如阿拉伯数字、英文字母和运算符等。由字母表中的字符，我们可以组成基本的单词，例如数 123 由 1、2 和 3 这三个数字构成，而标志符 abc 由 a、b 和 c 这 3 个字母组成。在编译领域，给单词起了一个专门的术语，叫 Token，其中包含了单词的类型和单词的值。例如 123 和 456 同样是数，但它们的值不一样；而数 123 和标志符 abc 是不同类型的单词。

识别了 token 之后，我们就可以按照上一节文法所规定的法则去组成合法的声明、表达式和语句等语法成份。下面我们来分析一下 UCC 源代码目录 ucc\examples\sc 中的代码，以下结构体来源于文件 lex.h，描述了与 Token 相关的类型和值，文件名中的 lex 是英文单词 lexical 的前缀。文件 ucc\examples\sc\tokens.txt 包含了各 token 类型。

```
typedef struct{
    TokenKind kind;
    Value value;
}Token;
```

而 ucc\examples\sc\lex.c 中的 GetToken() 函数则完成了对 Token 的识别。在图 1.5 第 6 至 8 行，我们先跳过空格、制表符、回车和换行等空白符，第 12 至 19 行完成了对标志符的识别。标志符以字母开头，后面紧跟若干个数字和字母。由于 if, else 和 while 等关键字也符合标志符的特征，我们需要在第 19 行调用 GetKeywordKind 函数，判断一下识别出来的标志符是不是关键字。标志符主要用于变量名和函数名。图 1.5 第 20 至 27 行实现的是对数 num 的识别，为简单起见，只完成了对十进制整数的识别。在 C 编译器中，我们要做得更复杂些，需要处理十进制整数、十六进制整数、float 和 double 浮点数等。当然，还有许多简单

的单词是由一个字符构成的，例如+、-、\*和/等运算符，在第 29 至 33 行主要是针对这些 token 的识别。这就是一个简单的用纯手工打造的词法分析器。翻开经典的龙书，我们发现词法分析这一章竟然占了大几十页的篇幅，引入了自动机和正则表达式等概念，用正则表达式来定义单词的构成规则，而用自动机来识别相应单词。实际上这些理论是为了实现词法分析器的自动构造。类似的，在文法的基础上构造语法分析器，也有自动和手工之分。龙书在语法分析这一章中引入的难懂众生的 LR 分析，实际上就是为了自动生成语法分析器。由于我们选择手工打造词法和语法分析器的技术路线，因此可以把自动机、正则表达式和 LR 分析等概念暂且搁置。

```

1 Token GetToken(void){
2     Token token;
3     int len = 0;
4     memset(&token, 0, sizeof(token));
5     // skip white space
6     while(IsWhiteSpace(curChar)){
7         curChar = NextChar();
8     }
9     TryAgain:
10    if(curChar == EOF_CH){
11        token.kind = TK_EOF;
12    }else if(isalpha(curChar)){//id or keyword
13        len = 0;
14        do{
15            token.value.name[len] = curChar;
16            curChar = NextChar();
17            len++;
18        }while(isalnum(curChar) && len < MAX_ID_LEN);
19        token.kind = GetKeywordKind(token.value.name);
20    }else if(isdigit(curChar)){//number
21        int numVal = 0;
22        token.kind = TK_NUM;
23        do{
24            numVal = numVal*10 + (curChar-'0');
25            curChar = NextChar();
26        }while(isdigit(curChar));
27        token.value.numVal = numVal;
28    }else{
29        token.kind = GetTokenKindOfChar(curChar);
30        if(token.kind != TK_NA){// '+', '-', '*', '/', ...
31            token.value.name[0] = curChar;
32            curChar = NextChar();
33        }else{
34            Error("illegal char \'%x\' .\n", curChar);
35            curChar = NextChar();
36            goto TryAgain;
37        }
38    }
39    return token;
40 }
```

图 1.5 GetToken()

下面我们以 PrimaryExpression 为例，讨论如何由产生式编写非终结符相应的分析函数。对非终结符 PrimaryExpression 而言，有 id、num 和(expression)这三个候选式。如果从输入

流中得到的当前 token 类型是左括号 TK\_LPAREN，则由图 1.6 的第 7 至 10 行去识别候选式 (expression)。通过第 8 行的 NEXT\_TOKEN 跳过左括号，读入下一个 token。查看 NEXT\_TOKEN 的宏定义，可以发现我们调用的正是前文所述的词法分析器 GetToken()。

```
#define NEXT_TOKEN do{curToken = GetToken();}while(0)
```

识别左括号之后，由产生式 PrimaryExpression 所制定的规则，我们预期接下来的 token 会构成一个合法的 Expression。而 Expression 是非终结符，我们也需要编写一个名为 Expression() 的函数来识别之。而在第 9 行只要调用函数 Expression() 即可，从 Expression() 函数返回后，在第 10 行，我们期望当前终结符是右括号。在函数 Expect() 中会进行比对，如果当前 token 不是右括号则报错；否则取下一个 token 作为当前 token。图 1.6 第 3 至 6 行用于处理标志符 id 和数 num。由于任何由 PrimaryExpression 生成的字符串，只能以标志符、数或者左括号开头，即 PrimaryExpression 的首符集是 {id, num, ()}，因此遇到的其他 token 都被视为非法的输入，我们会在第 12 行进行报错。同理，我们可以手工打造其他非终结符对应的分析函数。在文法中，我们用到了递归定义，而相应的分析函数就需要直接或间接递归调用。

```
// PrimaryExpression → id | num | (Expression)
1 static AstNodePtr PrimaryExpression(void) {
2     AstNodePtr expr = NULL;
3     if(curToken.kind == TK_ID || curToken.kind == TK_NUM) {
4         expr = CreateAstNode(curToken.kind, &curToken.value, NULL, NULL);
5         //取下一个 token
6         NEXT_TOKEN;
7     }else if(curToken.kind == TK_LPAREN) {
8         NEXT_TOKEN;
9         expr = Expression();
10        Expect(TK_RPAREN);
11    }else{
12        Error("expr: id or '(' expected.\n");
13    }
14    return expr;
15 }
```

图 1.6 PrimaryExpression()

在图 1.6 中第 4 行，我们调用函数 CreateAstNode() 创建了一个 astNode 结点，用于存储通过语法分析所得到的信息，例如遇到数 123 时，我们要记录在 astNode 结点中记录其类型为 TK\_NUM，值为 123。struct astNode 的定义如下所示。其中，op 域用于存放结点的类型，而 value 域代表了结点的值。而 kids[2] 域很自然地让我们联想到数据结构中的二叉树。对于 30 \* 50 这样的表达式，分析到 30 时，我们会为 30 构造一个 astNode 结点，遇到运算符 \* 时，也会为乘法运算符构造一个结点。显然乘法是个二元运算符，我们可用 kids[2] 来分别指向乘法运算符的左、右操作数。通过语法分析，我们最终可得到由许多 astNode 结点通过 kids[2] 指针相连构成的一棵树，这棵树被称为抽象语法树 (Abstract Syntax Tree)，缩写为 ast。抽象语法树与图 1.1 中的分析树是不同的。分析树中的内部结点由非终结符构成，而叶子结点由终结符构成，整棵分析树反映的是由文法开始符号进行推导的过程。实际上，我们没有必要去显式地构造分析树。由于每个非终结符与语法分析器中的一个函数相对应，如果把这些函数相互调用的关系画成一棵树，就可以得到分析树。分析树有时也被称为具体语法树，反映这棵树与产生式直接相关。而抽象语法树的内部结点通常为运算符，如前面的乘法运算符，叶子结点则是运算符所需要的操作数。此处“抽象”反映的是我们从 token 流中提取信息，构造适合进一步分析和处理的语法树的过程。

```
typedef struct astNode{
```

---

```

    TokenKind op;
    Value value;
    struct astNode * kids[2];
} * AstNodePtr;

```

接下来我们来看看如何处理由“若干个 PrimaryExpression() 相乘或相除”构成的乘法表达式。这里，我们把关注的焦点放在如何实现运算符的左结合和右结合上。例如对于表达式  $100/10/2$  而言，如果我们规定除法运算符是左结合的，由于词法分析是从左到右进行扫描的，按照图 1.7 的第 3 至 16 行的代码，我们可以构造一棵适合进行左结合运算的抽象语法树，分析完  $100/10$  时，就可以构造一个语法树，这棵树以除法运算符为根结点，操作数 100 和 10 对应的结点为左右子树。而接下来的 token 仍然是除法运算符，可把  $100/10$  对应的语法树作为左子树，用操作数 2 对应的结点作为右子树，去构造一棵新的以除法运算符为根结点的语法树，这个过程循环进行，因此在第 5 行我们用的是 while 循环。而如果规定除法运算是右结合的，则我们可按图 1.7 中的第 18 至 32 行的代码进行处理，与这部分代码对应的产生式如下所示：

```

MultiplicativeExpression → PrimaryExpression
MultiplicativeExpression → PrimaryExpression * MultiplicativeExpression
MultiplicativeExpression → PrimaryExpression / MultiplicativeExpression

```

这是右递归的产生式。对  $100/10/2$  来说，仍然是从左到右进行扫描的，但分析完  $100/10$  后，我们还不能为之构造语法树，我们要看一下 10 后面是否还有乘法或除法运算符，如果有，则运算数 10 要与其右侧的除法运算符先结合。因此先构造出来的语法树是  $10/2$ ，之后再以这棵子树为右子树，之前已经分析得到的 100 作为左子树，以除法运算法为根结点，构造一棵适合进行右结合运算的语法树。对候选式 PrimaryExpression / MultiplicativeExpression 的分析过程与前文对非终结符 PrimaryExpression 的分析过程类似，对于候选式中的非终结符，我们直接调用与之对应的同名函数来识别接下来的 token 流；对于候选式中的终结符，我们判断一下当前读入的 token 是否与之一致。图 1.7 中的第 19、20、26 和 28 行实际上就完成了这个分析过程，其余的几行代码用于创建语法树结点。

```

1 static AstNodePtr MultiplicativeExpression(void) {
2 #ifndef MUL_RIGHT_ASSOCIATE
3     AstNodePtr left;
4     left = PrimaryExpression();
5     while(curToken.kind == TK_MUL || curToken.kind == TK_DIV) {
6         Value value;
7         AstNodePtr expr;
8         memset(&value, 0, sizeof(value));
9         sprintf(value.name, MAX_ID_LEN, "t%d", NewTemp());
10        expr = CreateAstNode(curToken.kind, &value, NULL, NULL);
11        NEXT_TOKEN;
12        expr->kids[0] = left;
13        expr->kids[1] = PrimaryExpression();
14        left = expr;
15    }
16    return left;
17 #else
18     AstNodePtr left;
19     left = PrimaryExpression();
20     if(curToken.kind == TK_MUL || curToken.kind == TK_DIV) {
21         Value value;
22         AstNodePtr expr;
23         memset(&value, 0, sizeof(value));

```

```

24     sprintf(value.name,MAX_ID_LEN,"t%d",NewTemp());
25     expr = CreateAstNode(curToken.kind,&value,NULL,NULL);
26     NEXT_TOKEN;
27     expr->kids[0] = left;
28     expr->kids[1] = MultiplicativeExpression();
29     return expr;
30 }else{
31     return left;
32 }
33 #endif
34 }

```

图 1.7 MultiplicativeExpression()

对于  $100/10/2$  来说，由图 1.7 第 3 至 16 行，我们为其构造的语法树如图 1.8 左侧所示；而按照图 1.7 第 18 至 32 行所创建的语法树如图 1.8 右侧所示。

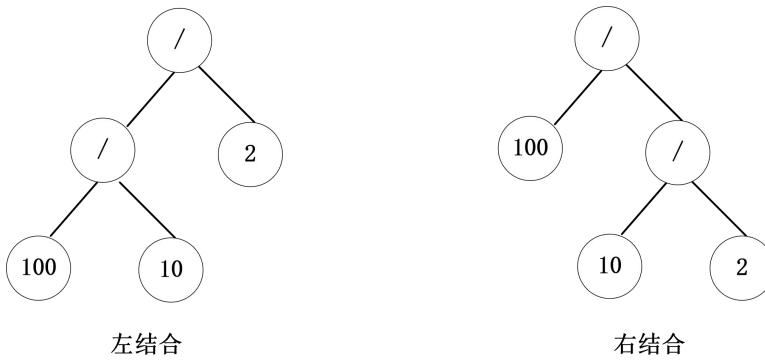


图 1.8 左结合和右结合的语法树

如果规定除法是右结合的，则  $100/10/2$  的运算结果为 20；而如果除法是左结合的，则  $100/10/2$  的结果是 5。定义一个新语言时，若硬要把除法规定为右结合，这是违反直觉和习惯的。我们只是以此为例来说明应如何处理右结合的运算符。在 C 语言中，表达式 “ $a = b = c$ ” 中的运算符 “=” 是右结合的。

在图 1.7 中第 9 行和第 24 行，我们还看到了一个名为 NewTemp() 的函数。对表达式  $a * b * c$  来说，如果乘法是左结合的，我们先进行的运算是  $a * b$ ，运算结果可存放到一个临时变量  $t1$  中，之后再用这个临时变量  $t1$  与  $c$  进行乘法运算，其结果再保存到另一个临时变量  $t2$  中，如下所示：

```

t1 = a * b
t2 = t1 * c

```

处理完乘除运算，我们再来讨论用于加减运算的函数 AdditiveExpression()，如图 1.9 所示，可以发现它与函数 MultiplicativeExpression() 的大部分代码都是相似的。它们都是二元运算符，只是运算符不同而已。在 C 语言中，还有许多类似的二元运算符，如果为每个这样的非终结符都构造一个独立的处理函数，则会出现大量的代码冗余。我们可以考虑把对二元运算符的处理归并到一个函数中进行统一处理。在后续章节分析 UCC 编译器的源代码时，我们会在 `ucc\src\expr.c` 中看到 ParseBinaryExpression() 这个函数，C 语言中所有的二元运算表达式都由这个函数进行分析。

```

1 static AstNodePtr AdditiveExpression(void) {
2 #ifndef ADD_RIGHT_ASSOCIATE
3     AstNodePtr left;
4     left = MultiplicativeExpression();
5     while(curToken.kind == TK_SUB || curToken.kind == TK_ADD) {
6         Value value;
7         AstNodePtr expr;

```

```

8     memset(&value, 0, sizeof(value));
9     sprintf(value.name, MAX_ID_LEN, "t%d", NewTemp());
10    expr = CreateAstNode(curToken.kind, &value, NULL, NULL);
11    NEXT_TOKEN;
12    expr->kids[0] = left;
13    expr->kids[1] = MultiplicativeExpression();
14    left = expr;
15 }
16 return left;
17 #else
18 AstNodePtr left;
19 left = MultiplicativeExpression();
20 if(curToken.kind == TK_SUB || curToken.kind == TK_ADD) {
21     Value value;
22     AstNodePtr expr;
23     memset(&value, 0, sizeof(value));
24     sprintf(value.name, MAX_ID_LEN, "t%d", NewTemp());
25     expr = CreateAstNode(curToken.kind, &value, NULL, NULL);
26     NEXT_TOKEN;
27     expr->kids[0] = left;
28     expr->kids[1] = AdditiveExpression();
29     return expr;
30 }else{
31     return left;
32 }
33 #endif
34 }

```

图 1.9 AdditiveExpression()

至此，我们已基本完成对加减乘除四则运算的分析，图 1.10 的右侧给出了分析完 $(a+b)*c$ 后，按左结合运算时，分析器为我们构造的语法树。图 1.10 左侧的分析树则反映了各分析函数之间的调用关系，其中标注的虚线反映了“在语法分析时创建语法树”的过程。

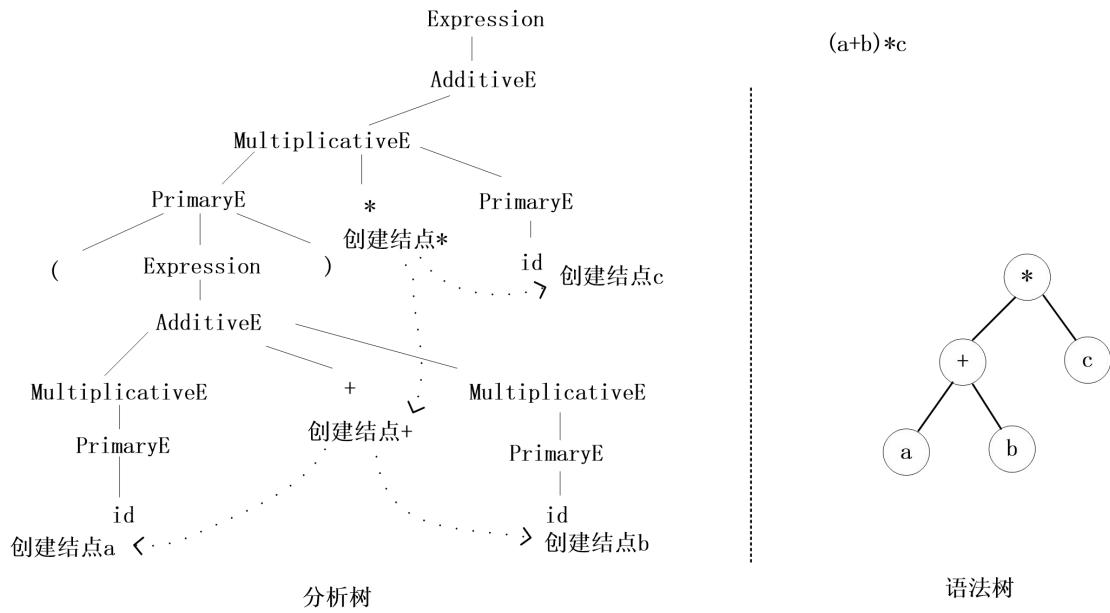


图 1.10 表达式对应的分析树和语法树

后续的语义分析和中间代码生成等动作，我们都会在语法树上进行。例如，图 1.11 中的函数 `VisitArithmeticNode()` 就是一个简单的中间代码生成器，其中的代码一定让我们条件

反射般地想起了当年在《数据结构》课中那熟悉的二叉树后序遍历。若对图 1.10 右侧的语法树进行后序遍历，可得到以下中间代码：

```
t0 = a + b
t1 = t0 * c
```

为了简单起见，在图 1.11 中，我们只是把中间代码以字符串的形式直接打印出来，而 UCC 编译器会将中间代码保存为四元式结构，之后再进行相应的优化。

```
1 static void Do_PrintNode(AstNodePtr pNode) {
2     if(pNode->op == TK_NUM){//打印整数
3         printf("%d ",pNode->value.numVal);
4     }else{//打印标志符
5         printf("%s ",pNode->value.name);
6     }
7 }
8 void VisitArithmeticNode(AstNodePtr pNode) {
9     if(pNode && IsArithmeticNode(pNode)){
10         VisitArithmeticNode(pNode->kids[0]);
11         VisitArithmeticNode(pNode->kids[1]);
12         if(pNode->kids[0] && pNode->kids[1]){//后序遍历
13             printf("\t%s = ", pNode->value.name);
14             Do_PrintNode(pNode->kids[0]);
15             printf("%s ",GetTokenName(pNode->op));
16             Do_PrintNode(pNode->kids[1]);
17             printf("\n");
18         }
19     }
20 }
```

图 1.11 VisitArithmeticNode()

### 1.3.2 声明

在这一小节中，我们要处理的是变量或函数的声明，相应的代码在 `ucc\examples\sc\decl.c` 中。为了使函数能有不同个数的参数，我们为图 1.4 中的文法增加了一个非终结符 `ParameterList`，实际运用的文法如下所示：

```
Declaration → int Declarator
Declarator → * Declarator | PostfixDeclarator
PostfixDeclarator → DirectDeclarator | PostfixDeclarator [num]
                    | PostfixDeclarator (ParameterListopt)
ParameterList → int | ParameterList , int
DirectDeclarator → id | (Declarator)
```

以“`int (*f(int,int,int))[4];`”这个看起来有点复杂的声明为例，在 Windows 或 Linux 平台上进入命令行，运行以下命令，我们得到的结果是“`f is: function(int,int,int) which returns pointer to array[4] of int`”。分析器告诉我们，`f` 实际上是一个带 3 个 `int` 参数的函数，其返回值是一个指针，指向包含 4 个整数的数组。

```
// Windows
D:\src\ucc\examples\sc> nmake -f Makefile.win
// Linux
iron@ubuntu:sc$ make
```

由于我们在这个简单的语言中只引入了 `int` 这个基本类型，因此所有的声明都是以 `int` 开始的，之后再跟上声明符 `Declarator`。在声明符 `Declarator` 中，我们可以进一步指定要声明的标志符为指针类型、数组类型或者函数类型。如果把`*`、`[]`和`()`看成类型运算符，把 `int`

这样的基本类型看成操作数，通过声明，我们实际上是在书写类型表达式。通过整数的四则运算表达式，我们希望 CPU 做计算，从而可以得到运算结果；而通过类型表达式，程序员为各个标志符指明了其类型。非终结符 Declarator 代表的实际上是以下集合：

```
{ PostfixDeclarator, *PostfixDeclarator,
  **PostfixDeclarator, *** PostfixDeclarator, ...}
```

而非终结符 PostfixDeclarator 代表的则是以下集合：

```
{
  DirectDeclarator, DirectDeclarator [num],
  DirectDeclarator (ParameterList), DirectDeclarator [num] [num],
  DirectDeclarator (ParameterList) (ParameterList),
  DirectDeclarator [num] (ParameterList),
  DirectDeclarator (ParameterList) [num], ...
}
```

在这里，我们可以发现 `int f(int)(int)` 竟然也是一个符合文法的声明，因为我们完全可以通过非终结符 Declarator 生成这个字符串。但 GCC、Visual Studio、Clang 和 UCC 都会给我们一个大大的错误提示，`int f(int)(int)` 是个非法的声明，该函数企图把函数作为返回值。在 UCC 给出的错误提示中的 “(declchk.c,629)”，表示我们是在 UCC 源代码文件 declchk.c 的第 629 行检查到了这个错误，这样的出错信息有助于对 UCC 编译器的跟踪和分析。

```
iron@ubuntu:test$ gcc hello.c
hello.c:2:5: error: 'f' declared as function returning a function
iron@ubuntu:test$ ucc hello.c
(declchk.c,629):(hello.c,2):error:function cannot return function type
ucc invoke command error:/home/iron/bin/ucl -o hello.s hello.i
iron@ubuntu:test$ clang hello.c
hello.c:2:6: error: function cannot return function type 'int (int)'
int f(int) (int);
^
1 error generated.
```

在标准 C 语言中，这个声明之所以“非法”，并不是因为其不遵守文法制定的规则，而是它不遵守语义规则。C 语言的标准文法规定了语法结构，但文法的能力是有限的，很多其他的限制和要求需要通过英语等自然语言来表达。C 语言附加的语义规则规定不能把函数作为返回值，请参见 ansi.c.txt 的 “3.5.4.3 Function declarators” 这一节。我们为了精准地刻画 C 语言这样的无穷集合，引入了文法，但最后却发现世界并不是那么完美，有许多的语义规则很难用我们现在介绍的文法来表达，或者即使能用文法来表达，也会把整个文法弄得非常复杂。实际上，即使是在文法中表达“函数调用时，实参数要和形参数一致”这样简单的语义规则，都要费一番周折。解决问题的思路可以有：

(1) 引入表达能力更强的文法。到目前为止，我们介绍的文法被称为上下文无关文法，(context-free grammar)，其所能表示的范围较小，而若要描述更大的范围，则需要能力更强的文法。关于这方面的研究属于“形式语言与自动机”范畴，沿着这个方向走下去，最终我们会膜拜图灵和丘奇等开天辟地的祖师爷。

(2) 用英语等自然语言来描述未在文法中体现的语义规则。就如企业面试有一面、二面、三面和四面一样，我们可以仍基于上下文无关文法进行语法分析，这相当于一面；通过后，再根据自然语言描述的语义规则去编写语义分析器，相当于作二面。这样的好处是为编程语言制定的标准文法简明易懂，但即使是同一篇文章，不同人读后都会有不同的感受，这就是所谓的 “There are a thousand Hamlets in a thousand people's eyes”。根据自然语言表述的语义规则，各大公司实现出来的语义分析器也会有些细微的差别。在后续章节讨论 UCC 编译器的语义检查时，我们会与 GCC、Clang 和 cl.exe 等常见编译器进行对比，届时可以发现各个

编译器的行为是存在一些差异的。实际的编译器几乎都是沿着(2)这个方向走的。如果有一天，再复杂的语义规则都可以形式化了，那按照“可以形式化，就可以自动化”的思路，程序员差不多就到了下岗的时候了。

言归正传，上述非终结符 Declaration 用于构造类型表达式，就如非终结符 Expression 用于刻画算术运算表达式一样。只不过 Declaration 中的运算符换成了\*、[]和(), 而操作数换成了 int 等基本类型。提到把类型作为参数，我们会联想到 C++ 语言中的模板。不论是类模板还是函数模板，都可以视为一个不完整的类型表达式，需要程序员提供类型作为操作数以构成一个完整的类型表达式。

以声明“int (\*f(int,int,int))[4];”为例，我们会为之造一棵形如图 1.12 右侧所示的语法树。我们在图 1.12 左侧画出了相应的分析树，其中的虚线标记了“在语法分析时构造语法树”的过程。

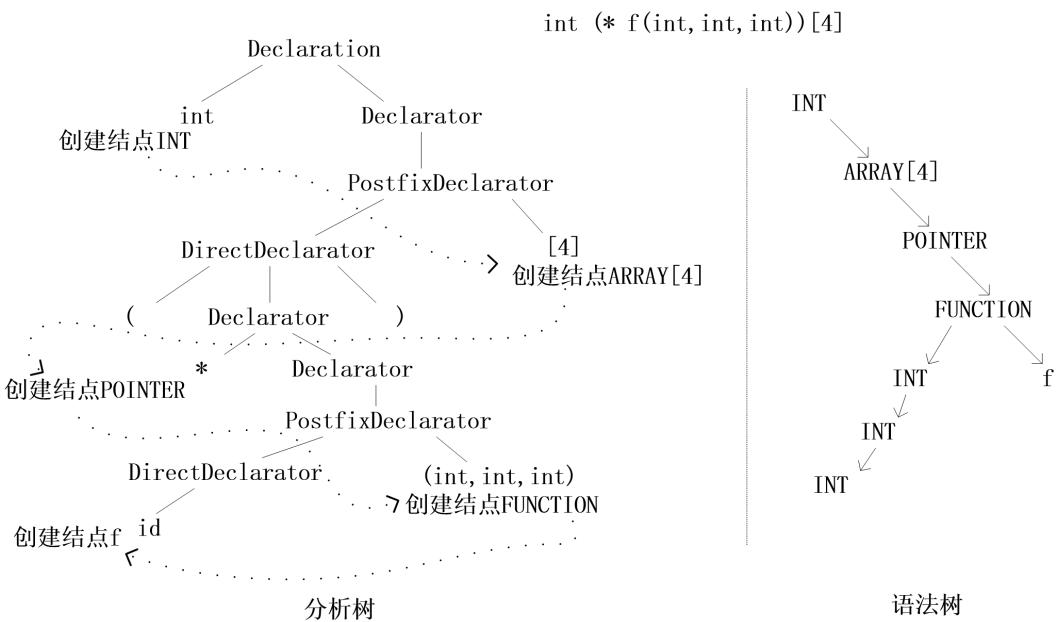


图 1.12 复杂声明的分析树与语法树

在图 1.12 的语法树中，我们用语法树结点的 `kids[1]` 域构成一个链表，来描述标志符 `f` 的类型信息。在 `FUNCTION` 结点中，我们用其 `kids[0]` 域来记录其形参列表。图 1.12 中语法树所代表的类型信息可简化为以下链表：

`INT` ---> `ARRAY[4]` ---> `POINTER` ---> `FUNCTION` ---> `f`

我们由链首出发，从左向右来构造一个新的类型，其过程如下所示：

- (1) `INT` 为基本类型，所得类型记为 `int`。
- (2) 把类型运算符 `ARRAY[4]` 作用于上一步的结果，所得类型为 `array[4] of int`。
- (3) 把类型运算符 `POINTER` 作用于上一步的结果，所得类型为 `pointer to array[4] of int`。
- (4) 把类型运算符 `FUNCTION` 作用于上一步的结果，即把上一步的类型作为函数返回值的类型，而函数的形参列表(`int,int,int`)则存于 `FUNCTION` 结点的 `kids[0]` 域中，所得类型为 `function (int,int,int) which returns pointer to array[4] of int`。
- (5) 上一步所得的类型即为标志符 `f` 的类型，记为“`f is: function(int,int,int) which returns pointer to array[4] of int`”。

如果要把 `f` 的类型信息打印出来，我们需要先打印字符串“`f is :`”，而 `f` 所对应的结点是位于上述单链表的末尾。而单链表的扫描需要从链首开始的，这里我们可用函数递归调用来实现单向链表的“逆向遍历”。由如图 1.13 第 4 行可知，只有遍历了当前结点的后继后，才

能访问当前结点，由此实现了单链表的“逆向遍历”。之后的第 5 至 32 行根据当前结点的类型，打印出相应的类型信息。

```

1 void VisitDeclarationNode(AstNodePtr pNode) {
2     AstNodePtr curParam;
3     if(pNode) {
4         VisitDeclarationNode(pNode->kids[1]);
5         switch(pNode->op) {
6             case TK_POINTER:
7                 printf("pointer to ");
8                 break;
9             case TK_FUNCTION:
10                printf("function()");
11                curParam = pNode->kids[0];
12                while(curParam) {
13                    printf("%s", curParam->value.name);
14                    curParam = curParam->kids[0];
15                    if(curParam) {
16                        printf(",");
17                    }
18                }
19                printf(" which returns ");
20                break;
21            case TK_ARRAY:
22                printf("array[%d] of ", pNode->value.numVal);
23                break;
24            case TK_ID:
25                printf("%s is: ", pNode->value.name);
26                break;
27            case TK_INT:
28                printf("int \n");
29                break;
30            default:
31                printf("unknown: %s", pNode->value.name);
32        }
33    }
34 }
```

图 1.13 输出类型信息

对二维数组的声明“`int arr[3][5];`”来说，我们会为之造一棵如图 1.14 右侧所示的语法树，左侧分析树中的虚线标注了在“语法分析时构造右侧语法树”的过程。

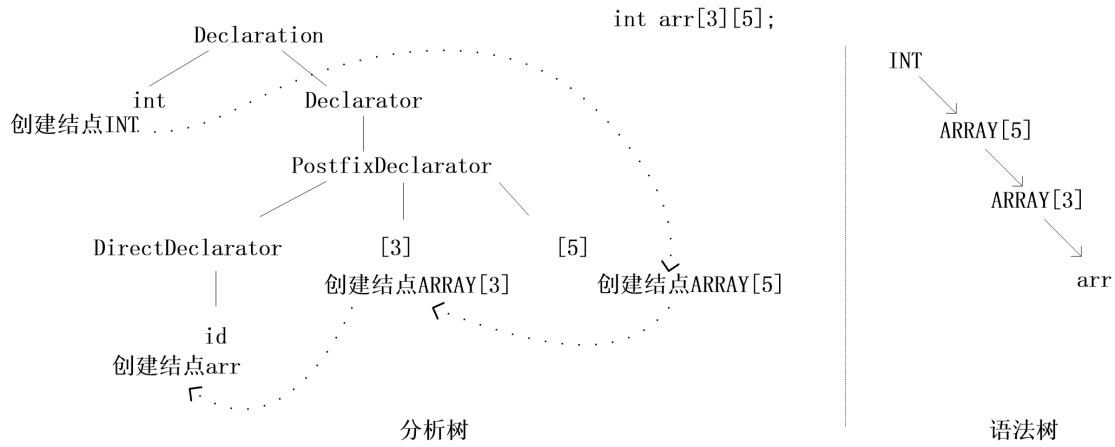


图 1.14 二维数组对应的分析树与语法树

由图 1.14 左侧的分析树可知, 当 Declaration() 函数返回后, 所得到的语法树(此处实际上是个单链表)中就记录了标志符 arr 的完整类型信息。我们仍然由链首出发, 从左向右来构造一个新的类型, 其过程如下所示:

- (1) INT 为基本类型, 所得类型记为 int。
- (2) 把类型运算符 ARRAY[5] 作用于上一步的结果, 所得类型为 array[5] of int。
- (3) 把 ARRAY[3] 类型运算符作用于上一步的结果, 所得类型可以记为 array[3] of array[5] of int。
- (4) 上一步所得的类型即为标志符 arr 的类型, 记为 “arr is: array[3] of array[5] of int”。

图 1.15 给出了非终结符 PostfixDeclarator 对应的分析函数, 第 4 至 13 行用于分析 [num], 图 1.14 语义树中的 ARRAY 结点就是在图 1.15 第 6 行创建的。而第 14 至 27 行用于分析 (ParameterList<sub>opt</sub>), 图 1.12 语义树中的 FUNCTION 结点由图 1.15 第 17 行构造。

```

1 static AstNodePtr PostfixDeclarator(void) {
2     AstNodePtr decl = DirectDeclarator();
3     while(1) {
4         if(curToken.kind == TK_LBRACKET) { // [num]
5             NEXT_TOKEN;
6             decl = CreateAstNode(TK_ARRAY, &curToken.value, NULL, decl);
7             if(curToken.kind == TK_NUM) {
8                 decl->value.numVal = curToken.value.numVal;
9                 NEXT_TOKEN;
10            } else{
11                decl->value.numVal = 0;
12            }
13            Expect(TK_RBRACKET);
14        } else if(curToken.kind == TK_LPAREN) { // (ParameterListopt)
15            AstNodePtr * param;
16            NEXT_TOKEN;
17            decl = CreateAstNode(TK_FUNCTION, &curToken.value, NULL, decl);
18            param = &(decl->kids[0]);
19            while(curToken.kind == TK_INT) {
20                *param = CreateAstNode(TK_INT, &curToken.value, NULL, NULL);
21                param = &((*param)->kids[0]);
22                NEXT_TOKEN;
23                if(curToken.kind == TK_COMMA) {
24                    NEXT_TOKEN;
25                }
26            }
}

```

```

27         Expect (TK_RPAREN);
28     }else{
29         break;
30     }
31 }
32 return decl;
33 }

```

图 1.15 PostfixDeclarator()

而非终结符 Declarator、DirectDeclarator 和 Declaration 的分析函数，与 PostfixDeclarator 类似，就不再赘述。观察 Declaration 的产生式，我们可以发现这样的产生式一次只能声明一个标志符，无法同时声明多个标志符，例如“int a, b;”。

Declaration → int Declarator

这个问题不难解决，把上述产生式改为如下即可：

Declaration → int DeclaratorList

DeclaratorList → Declarator | DeclaratorList , Declarator

从上面的产生式，我们也能很好地理解在下面的代码中，为什么 c 是指针，而 d 只是 int。从语法结构上，“\* c”是由一个声明符 Declarator 生成，而“d”则由另一个声明符 Declarator 生成。

```
int * c, d;
```

当然在前文中我们举了“int (\*f(int,int,int))[4];”这个例子，这样复杂的声明难不倒编译器，但会难倒不少程序员，更好的编程风格是使用 typedef，使得类型信息清晰可见。大部分程序员是习惯从左到右来阅读源代码的，但是形如“int (\*f(int,int,int))[4];”这样的复杂声明，需要先看 f(int,int,int)，再向外看。这不符合人的直觉。因此，还是要写成以下这样符合从左向右阅读习惯的代码，毕竟代码是写给大家看的。

```

typedef int ARRAY[4];
ARRAY * f(int, int ,int);

```

### 1.3.3 语句

理解了变量和函数的声明，及如何处理四则运算表达式后，接下来我们要在表达式的基础上去实现控制流语句，例如 if 语句和 while 语句。与语句相关的产生式如下所示：

```

Statement → IfStatement | WhileStatement
           | CompoundStatement | ExpressionStatement
IfStatement → if ( Expression) Statement
IfStatement → if ( Expression) Statement else Statement
WhileStatement → while(Expression) Statement
CompoundStatement → { StatementListopt}
StatementList → Statement | StatementList Statement
ExpressionStatement → id = Expression ;
ExpressionStatement → Declaration ;

```

根据上述文法，图 1.16 是我们为 ExpressionStatement 所编写的分析函数，之前我们已经实现了表达式 Expression 和声明 Declaration 对应的分析函数。依照 C 语言标准，此处我们把“id = Expression”视为赋值表达式，“赋值表达式之后再跟上分号”就构成了表达式语句 ExpressionStatement，图中第 3 至 14 行实现了对赋值表达式的识别。为简单起见，我们把“Declaration ;”也视为表达式语句，图中第 15 至 20 行用于对其进行语法分析。第 15 行的 IS\_PREFIX\_OF\_DECL() 就是我们前文介绍的首符集的概念，只有当前 token 属于非终结符 Declaration 的首符集时，我们才会执行 17 至 20 行的代码去识别“Declaration ;”。这里可能存在问题是，如果第 2 行的 TK\_ID 也属于 Declaration 的首符集时该如何处理。就跟在

陌生地方遇到叉路口一样，如果向左向右傻傻分不清楚时，不妨先右转，发现此路不通时，再退回到叉路口左转。在后续章节分析 UCC 源代码时，我们会在 `ucc\ucl\lex.c` 中看到 `BeginPeekToken()` 和 `EndPeekToken()` 这一对函数，它们就是用来做尝试和回溯的。当然在遇到较简单的文法时，也可以对文法进行改造，进行“提取左公因子”的操作。

```

1 static AstStmtNodePtr ExpressionStatement(void) {
2     if(curToken.kind == TK_ID) {
3         // id = Expression;
4         AstStmtNodePtr assign = CreateStmtNode(TK_ASSIGN);
5         assign->kids[0] = CreateAstNode(TK_ID, &curToken.value, NULL, NULL);
6         NEXT_TOKEN;
7         if(curToken.kind == TK_ASSIGN) {
8             NEXT_TOKEN;
9             assign->expr = Expression();
10        } else{
11            Error("stmt:      '=' expected.\n");
12        }
13        Expect(TK_SEMICOLON);
14        return assign;
15    } else if(IS_PREFIX_OF_DECL(curToken.kind)) {
16        // Declaration ;
17        AstStmtNodePtr decl = CreateStmtNode(TK_DECLARATION);
18        decl->expr = Declaration();
19        Expect(TK_SEMICOLON);
20        return decl;
21    } else{
22        Error("stmt:      id expected.\n");
23        return NULL;
24    }
25 }
```

图 1.16 ExpressionStatement()

在图 1.16 的第 4 和第 5 行，我们还看到了 `CreateStmtNode()` 和 `CreateAstNode()` 这两个不同的函数调用，它们都用于创建一个语法树结点。函数 `CreateStmtNode()` 创建的是 `astStmtNode` 对象，而 `CreateAstNode()` 创建的是 `astNode` 对象。对比以下结构体定义可以发现，`astStmtNode` 对象起始部分的数据恰好可以当作一个 `astNode` 对象来用，用面向对象语言的话来说，这就相当于结构体 `astStmtNode` 继承了结构体 `astNode`。这种技巧实际上也是用 C 来实现面向对象思想的常用方法。由于我们需要把 `if` 和 `while` 语句的相关信息也存储到抽象语法树的结点中，这需要更多的数据域，因此我们引入了 `astStmtNode` 结构体。而该结构体里的 `thenStmt` 等成员的作用，在讨论 `IfStatement` 和 `WhileStatement` 的分析函数时，就会一目了然。需要补充说明的是，我们在结构体 `astNode` 中，用词法单元的类型 `TokenKind` 来区别不同的抽象语法树结点，这么做的目的只是为了简单起见。在 UCC 的源代码中，我们需要专门为抽象语法树结点定义不同的结点类型，可参见 `ucc\ucl\opinfo.h`。即使在本节讨论的这么简单的语言中，我们也可以发现用 `TokenKind` 来区别不同的语法树是不够的。以乘法运算符`*`为例，该符号用于四则运算表达式“(a+b)\*c”时充当乘号；而在“int \* a;”中则不是作为乘号来使用。因此，在 `ucc\examples\sc\tokens.txt` 中，我们引入了 `TK_MUL` 和 `TK_POINTER`，用于区别运算符`*`是作乘号还是作指针之用。这实际上就是 C 语言的运算符重载。C++ 把运算符重载的能力提供给了程序员，以期获得更符合书写习惯和可读性更好的代码。

```

typedef struct astNode{
    TokenKind op;
```

```

    Value value;
    struct astNode * kids[2];
} * AstNodePtr;

typedef struct astStmtNode{
    TokenKind op;
    Value value;
    struct astNode * kids[2];
    /////////////////////////////////
    struct astNode * expr;
    struct astStmtNode * thenStmt;
    struct astStmtNode * elseStmt;
    struct astStmtNode * next;
} * AstStmtNodePtr;

```

表达式语句是我们的基本语句。而语句是递归定义的，就如任何递归调用的函数需要递归出口一样，基本语句就充当了递归出口的作用。同理，如果语句列表只用以下产生式来定义：

StatementList → StatementList Statement

而没有产生式“StatementList → Statement”作为递归出口，则 StatementList 会陷入死循环，是一个不正确的产生式。

下面，我们来讨论应如何翻译 if 语句。If 语句可分为以下两种，我们以带 else 分支的 if 语句为例来进行讨论。

IfStatement → if ( Expression) Statement else Statement  
 IfStatement → if ( Expression) Statement

以下就是一条简单的包含 else 的 if 语句，在中间代码层次，通过有条件的跳转和无条件的跳转，我们可以实现控制流的转移。而在 CPU 的指令集中，会有相应的硬件指令来实现有条件或无条件的跳转。生成实际汇编指令的工作，我们会在后续章节进行讨论。

```

if(c){
    a = f;
}else{
    b = k;
}

```

我们期望把上述 if 语句翻译为以下中间代码，其中的标号 Label\_0 和 Label\_1 用作 goto 的跳转目标。结构化程序设计只是让 C 程序员尽量少用 goto 语句，但到了中间代码或汇编代码层次，我们还是要通过“有条件或无条件跳转指令”来实现高级语言里的 if 和 while 等控制流语句。对于 if 语句，我们就需要建立一个语法树结点来存放“if 语句的表达式”、“then 语句”、“else 语句及其对应的标号 Label\_0”和“if 语句的下一条语句对应的标号 Label\_1”。

```

if(!c) goto Label_0 //表达式 c
a = f                //then 语句， 存于 astStmtNode 对象的 thenStmt 域
goto Label_1          //
Label_0:             //Label_0 是 else 语句的标号，存于 kids[0]
b = k                //else 语句， 存于 astStmtNode 对象的 elseStmt 域
Label_1:             //Label_1 是下一条语句的标号，存于 kids[1]

```

有了这样的直观体会后，我们再来看一下图 1.17，第 4 行创建了一个新的语法树结点，第 7 行用于记录 if 语句的表达式，第 10 行用于记录 then 语句的语法子树，第 12 至 17 行用于处理 else 分支。图 1.17 第 11 行调用的 CreateLabelNode() 函数用于创建一个新的标号。形如 Label\_0 和 Label\_1 的标号主要是给汇编器看的，而不是给程序员看的，我们不必像手工编写的汇编代码那样，需要去考虑取有意义的标号名。

1 static AstStmtNodePtr IfStatement(void) {

```

2     AstStmtNodePtr ifStmt = NULL;
3
4     ifStmt = CreateStmtNode(TK_IF);
5     Expect(TK_IF);
6     Expect(TK_LPAREN);
7     ifStmt->expr = Expression();
8     Expect(TK_RPAREN);
9
10    ifStmt->thenStmt = Statement();
11    ifStmt->kids[0] = CreateLabelNode();
12    if(curToken.kind == TK_ELSE){
13        NEXT_TOKEN;
14        ifStmt->elseStmt = Statement();
15        // label for the statement after if-statement
16        ifStmt->kids[1] = CreateLabelNode();
17    }
18
19    return ifStmt;
20
21 }

```

图 1.17 IfStatement()

在 if 语句的基础上，我们再来看看 while 语句。其实 while 语句可以看成是带 goto 的 if 语句。只不过以下中间代码中的“goto Label\_2”需要由编译器产生而已。

```

while(c) {
    a = f;
}

//对应的中间代码

Label_2:           //while 语句的标号，存于 kids[0]
if(!c) goto Label_3 //while 语句的表达式 c
a = f               //then 语句， 存于 astStmtNode 对象的 thenStmt 域
goto Label_2        //

Label_3:           //while 语句的下一条语句的标号，存于 kids[1]

```

While 语句对应的分析函数如图 1.18 第 1 至 14 行所示，第 3 行创建了一个 astStmtNode 结点，第 5 行在 kids[0]域中记录了形如 while 语句的标号，第 8 行记录 while 语句的表达式，而第 10 行用于记录 then 语句对应的语法子树，第 12 行在 kids[1]域中记录了 while 语句的下一条语句的标号。

```

1 static AstStmtNodePtr WhileStatement(void){
2     AstStmtNodePtr whileStmt = NULL;
3     whileStmt = CreateStmtNode(TK WHILE);
4
5     whileStmt->kids[0] = CreateLabelNode();
6     Expect(TK WHILE);
7     Expect(TK_LPAREN);
8     whileStmt->expr = Expression();
9     Expect(TK_RPAREN);
10    whileStmt->thenStmt = Statement();
11    // lable for the statement after while
12    whileStmt->kids[1] = CreateLabelNode();
13    return whileStmt;
14 }
15 // comStmt->next ----> list of statements
16 AstStmtNodePtr CompoundStatement(void){

```

```

17 AstStmtNodePtr comStmt;
18 AstStmtNodePtr * pStmt;
19 Value value;
20
21 comStmt = CreateStmtNode(TK_COMPOUND);
22 pStmt = &(comStmt->next);
23
24 Expect(TK_LBRACE);
25 while(isPrefixOfStatement(curToken.kind)) {
26     *pStmt = Statement();
27     pStmt = &((*pStmt)->next);
28 }
29 Expect(TK_RBRACE);
30 return comStmt;
31 }

```

图 1.18 While 和复合语句

而图 1.18 第 16 至 31 行用于分析复合语句，即由一对大括号包围的若干条语句。为了记录若干条语句构成的链表，我们在结构体 `astStmtNode` 中引入了 `next` 域，图 1.18 中第 22 和第 27 行用于构造单向链表，而在第 25 行中，我们要判断当前 token 是否为语句的首符。

熟悉了结构体 `astStmtNode` 中各成员域的作用后，我们可以来看看如何遍历 `Statement` 对应的语法树，从而产生上述中间代码。图 1.19 给出了用于为语句生成中间代码的函数 `VisitStatementNode()`，第 6 至 22 行用于处理 if 语句对应的语法树结点，第 23 至 31 行用于处理 while 语句对应的语句结点，而第 32 至 37 行则用于处理复合语句。

```

1 void VisitStatementNode(AstStmtNodePtr stmt) {
2     if(!stmt){
3         return;
4     }
5     switch(stmt->op) {
6         case TK_IF://为 if 语句产生中间代码
7             VisitArithmeticNode(stmt->expr);
8             if(stmt->kids[0] && stmt->kids[1]) {
9                 printf("\tif(!%s) goto %s \n",
10                     stmt->expr->value.name,stmt->kids[0]->value.name);
11                 VisitStatementNode(stmt->thenStmt);
12                 printf("\tgoto %s \n",stmt->kids[1]->value.name);
13                 printf("%s:\n",stmt->kids[0]->value.name);
14                 VisitStatementNode(stmt->elseStmt);
15                 printf("%s:\n",stmt->kids[1]->value.name);
16             }else{
17                 printf("\tif(!%s) goto %s \n",
18                     stmt->expr->value.name,stmt->kids[0]->value.name);
19                 VisitStatementNode(stmt->thenStmt);
20                 printf("%s:\n",stmt->kids[0]->value.name);
21             }
22             break;
23         case TK WHILE://为 while 语句产生中间代码
24             printf("%s:\n",stmt->kids[0]->value.name);
25             VisitArithmeticNode(stmt->expr);
26             printf("\tif(!%s) goto %s \n",
27                 stmt->expr->value.name,stmt->kids[1]->value.name);
28             VisitStatementNode(stmt->thenStmt);
29             printf("\tgoto %s \n",stmt->kids[0]->value.name);

```

```

30     printf("%s:\n",stmt->kids[1]->value.name);
31     break;
32 case TK_COMPOUND://为复合语句产生中间代码
33     while(stmt->next){
34         VisitStatementNode(stmt->next);
35         stmt = stmt->next;
36     }
37     break;
38 // 略
39 }

```

图 1.19 VisitStatementNode()

到此，我们对表达式、声明和语句的翻译应有了最直观的体会，现在可以看看语法分析器的 main() 函数了，如图 1.20 第 21 至 29 行所示。图 1.20 第 2 行的函数 NextCharFromMem() 用于从内存中读取一个字符，而第 12 行的函数 NextCharFromStdin() 则用于从当前进程的标准输入中读取一个字符。在图 1.20 第 23 行，我们调用了函数 InitLexer(NextCharFromStdin) 来初始化词法分析器，使之通过标准输入来读取下一个字符。

```

1 static const char * srcCode = "{int (*f(int,int,int))[4];}";
2 static char NextCharFromMem(void){
3     int ch = *srcCode;
4     srcCode++;
5     if(ch == 0){
6         return (char)EOF_CH;
7     }else{
8         return (char)ch;
9     }
10 }
11
12 static char NextCharFromStdin(void){
13     int ch = fgetc(stdin);
14     if(ch == EOF){
15         return (char)EOF_CH;
16     }else{
17         return (char)ch;
18     }
19 }
20
21 int main(int argc,char * argv[]){
22     AstStmtNodePtr stmt = NULL;
23     InitLexer(NextCharFromStdin);
24     NEXT_TOKEN;
25     stmt = CompoundStatement();
26     Expect(TK_EOF);
27     VisitStatementNode(stmt);
28     return 0;
29 }

```

图 1.20 main()

函数 InitLexer() 的代码在 ucc\examples\sc\lex.c 文件中，如下所示，其中 NextChar 是函数指针，其类型为 NEXT\_CHAR\_FUNC。

```

typedef char (* NEXT_CHAR_FUNC)(void);
static char curChar = ' ';
static NEXT_CHAR_FUNC NextChar;

```

```

void InitLexer(NEXT_CHAR_FUNC next) {
    if(next) {
        NextChar = next;
    }
}
Token GetToken(void) {
    //略
    curChar = NextChar();
    //略
}

```

不用改动 lex.c 中的代码，只要通过给 InitLexer() 传递不同的函数指针，我们就可以让 ucc\examples\sc\lex.c 中的词法分析器 GetToken() 从内存或从标准输入中读取下一个字符，这和面向对象语言中的多态异曲同工。实际上，C++ 的虚函数机制就是通过虚函数表来实现的，而虚函数表由若干个函数指针构成的。只要浏览一下 Linux 的内核源代码，我们就可以找到很多用于描述“虚函数表”的结构体，比如 struct file\_operations。Linux 内核虽然主要是用 C 语言实现，但我们还是能用 C 语言实现继承和多态这样的面向对象思想。只是创建虚函数表这样的操作需要 C 程序员自己动手，没有 C++ 编译器代劳而已。

我们要识别的是由非终结符 Program 产生的字符串，但在图 1.20 第 25 行我们调用的是 CompoundStatement()，这是因为 Program 的产生式如下所示，为简单起见，我们就不再为 Program 构造一个分析函数了。整个 Program 识别成功后，我们在图 1.20 第 26 行应该遇到文件结束符 EOF，第 27 行调用函数 VisitStatementNode 来遍历语法树，从而产生中间代码。

Program → CompoundStatement

## 1.4 UCC 编译器预览

### 1.4.1 UCC 的使用

通过第 1.3 节的例子 ucc\examples\sc，我们对如何根据语言的文法来编写语法分析器，在分析时构建语法树，之后在语法树的基础上产生中间代码有了一个感性的直观认识。当然，光有这些中间代码，C 程序员还不能得到相应的计算结果。C 编译器还要由中间代码产生汇编代码。在剖析 UCC 编译器前，让我们先熟悉一下 UCC 编译器的使用。UCC 编译器的大部分代码都是用标准 C 语言并调用 C 标准库来编写，可在 Linux 或 Windows 平台上生成 32 位的 x86 汇编代码，这些代码需要 32 位函数库的支持才能在相应系统中运行。在后续的章节中，为节省篇幅，我们主要以 32 位 Ubuntu 系统为例来讨论。在物理机上或者在 VMware 等虚拟机上安装 32 位 Ubuntu 系统的过程并不会太复杂，这里就不再画蛇添足。我们需要在 ucc/makefile 第 1 行和 ucc/driver/linux.c 第 7 行配置 UCC 的安装目录 UCCDIR，比如

“/home/iron/bin”。如果 UCC 的源代码被解压到目录 /home/iron/src/ucc，则经过以下步骤就可构建并安装 UCC 到目录 /home/iron/bin 中。

```

iron@ubuntu:ucc$ pwd
/home/iron/src/ucc
iron@ubuntu:ucc$ make -s
iron@ubuntu:ucc$ make -s install
iron@ubuntu:ucc$ make -s test

```

为了使用户能方便地使用 ucc 命令，我们还需要设置一下环境变量 PATH，具体的操作如下所示，由 “cd ~” 进入当前用户主目录，在 gedit 打开的 .bashrc 文件末尾添加一行 “export PATH=\$PATH:/home/iron/bin”，其中 “/home/iron/bin” 就是前文所设定的 UCCDIR。保存后

退出，重新打开一个终端，即可使用 ucc 命令。

```
iron@ubuntu:ucc$ cd ~
iron@ubuntu:~$ gedit .bashrc
```

接下来，我们用一个简单的例子来解释一下 UCC 的大致工作流程。编写以下 C 代码，存为文件 hello.c。这份代码用于求阶乘，其中有 if 语句、while 语句、库函数 printf 及递归函数 f。

```
#include <stdio.h>
int f(int n){
    if(n < 1){
        return 1;
    }else{
        return n * f(n-1);
    }
}
int main(int argc,char * argv[]){
    int i = 1;
    while(i <= 10){
        printf("f(%d) = %d\n",i,f(i));
        i++;
    }
    return 0;
}
```

C 源代码 hello.c 需要先经过预处理器(C PreProcessor)的预处理。预处理器会根据预设的 include 目录去查找并包含头文件，并对宏定义进行展开，如果找不到对应的头文件则报错，这类错误是预处理器报的错，还未到编译器阶段。

预处理器后的结果 hello.i，才是作为 C 编译器的输入。有时侯，C 编译器可能报出数以百计的语法错误，其原因可能宏定义时出了点差错，这时打开预处理后的文件看看，就能很清楚哪里出问题了。

编译器会对 hello.i 进行词法分析、语法分析、语义检查和中间代码生成，经过前面几节的准备，我们对这些概念应比较熟悉了，这几个阶段被称为编译器的前端，它们与具体的机器无关。C 编译器再根据中间代码生成不同硬件平台的汇编语言，这部分工作被称为编译器的后端，与具体的机器相关，不同机器的机器指令是各不相同的。当然，编译器还有“优化”这样的重点戏需要完成，这也是编译相关研究的热点。而中间代码实际上起到了连结前端和后端的桥梁作用。

在编译器生成汇编代码 hello.s 后，还需要借助汇编器 (assembler) 根据汇编语言来“装配”成机器码，由此产生了目标代码 hello.o，文件名后缀中的字母 “o” 是 object 的缩写。既然不同硬件平台的机器代码是不同的，如果定义一套中间代码，我们假设有一个虚拟的机器，其机器代码正好就是我们的中间代码。这样，程序员所编写的高级语言先被编译成中间代码，再把这些中间代码送给用软件实现的虚拟机来解释执行，各个平台上预先写好各自的中间代码虚拟机，那我们的中间代码就可以跨平台运行了。这一定让我们想起了 Java 和“Write Once , Run Anywhere”那让人热血沸腾的 Slogan。当然，与运行平台相关的工作及优化的重头戏就交给了 Java 虚拟机。Java 得到跨平台的代价是牺牲了一部分的运行效率，但在程序员比 CPU 和内存贵的今天，这种牺牲还是有经济上的意义的。一般而言，生成中间代码之后解释执行，比生成本地机器代码之后直接运行的效率更低，其原因就好比有一本英语原版书，翻译的方法可以有口译和笔译，若用口译的方式，每次我们都要找口译员解释一下，之后我们可能就忘了，下回再看时，可能还要劳烦别人再口译一次；而笔译的好处是笔译员翻译一遍后，我们就有了一份中文版的书，以后就不用再麻烦别人了。Java 虚拟机采取的加速

方案有即时编译（Just In Time），如果在运行时发现有些中间代码要被多次解释执行，那我们干脆就在动态运行时，把相应的中间代码翻译成机器代码，这就是所谓的“即时”。当然，道理很简单，做起来很难。

把目标模块 hello.o、函数库和其他的目标代码组装到一起，才能得到可执行程序 hello，这个工作被称为连接（Linking），也有写为链接的，由连接器（Linker）完成。在此阶段出现的错误有“全局变量(或函数)重复定义”和“全局变量(或函数)未定义”等。

图 1.21 给出了由 C 源程序 hello.c 得到可执行程序 hello 的主要过程，其中 hello.i 是预处理后的结果，hello.s 是汇编代码，hello.o 是目标代码。可以发现，要由 C 源代码得到一份可执行程序，我们需要预处理器、编译器、汇编器和连接器等诸多工具的配合，这实际上是一套工具链，平时我们口头上讲“使用 GCC 编译器”，实际上，我们是用这套工具链中的编译器来作为整个工具链的代表了。UCC 实现的是 C 编译器，而预处理器、汇编器和连接器仍然用 Linux 或 Windows 平台上的相应工具。

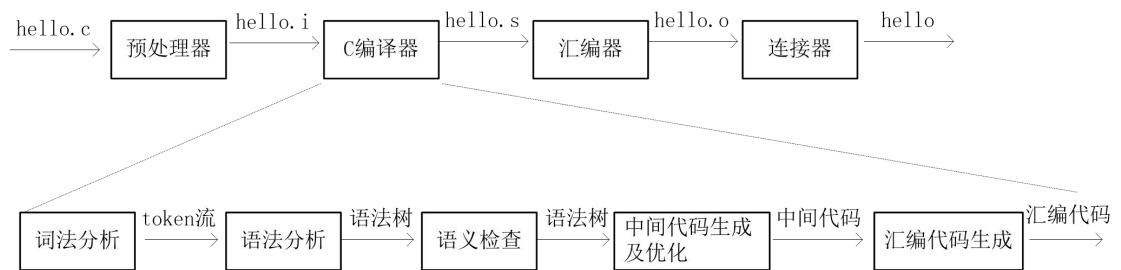


图 1.21 编译工具链

可通过以下操作来验证上述过程，其中用到了 ucc 这个命令，通过给 ucc 传递“-E”、“-S”和“-c”等不同参数来通知 ucc 调用预处理器、编译器和汇编器等不同工具。

```

iron@ubuntu:demo$ ls
build.bat hello.c Makefile
iron@ubuntu:demo$ ucc -E hello.c -o hello.i
iron@ubuntu:demo$ ls
build.bat hello.c hello.i Makefile
iron@ubuntu:demo$ ucc --dump-ast --dump-IR -S hello.i -o hello.s
iron@ubuntu:demo$ ls
build.bat hello.ast hello.c hello.i hello.s hello.ul Makefile
iron@ubuntu:demo$ ucc -c hello.s -o hello.o
iron@ubuntu:demo$ ls
build.bat hello.ast hello.c hello.i hello.o hello.ul Makefile
iron@ubuntu:demo$ ucc hello.o -o hello
iron@ubuntu:demo$ ls
build.bat hello.ast hello.i hello.s Makefile
hello hello.c hello.o hello.ul
iron@ubuntu:demo$ ./hello
f(1) = 1
f(2) = 2
f(3) = 6
f(4) = 24
f(5) = 120
f(6) = 720
f(7) = 5040
f(8) = 40320
f(9) = 362880
f(10) = 3628800

```

在 UCC 编译器的阅读和分析时，比较有用的参数还有用于输出语义树的“--dump-ast”，

和用于输出中间代码的“--dump-IR”，其结果分别是前文的 hello.ast 和 hello.uil。限于篇幅，我们就不列出 hello.i、hello.ast、hello.uil 和 hello.s 这些文本文件的内容，但结合 hello.c 去浏览一下这些文件，会对编译工具链的工作流程会有更加具体的认识。

实际上，上述的 ucc 命令只是编译器驱动（driver），用来驱动编译工具链的预处理器、编译器、汇编器和连接器。UCC 驱动的代码在 ucc\driver 目录中，而我们要剖析的 UCC 编译器的源代码则在目录 ucc\ucl 中。按“扫外围，剪裙边”的思路，下一小节，我们会先剖析一下 UCC 编译器驱动的代码。

## 1.4.2 UCC 驱动

在上一小节，通过以下四条命令，我们有意地给 ucc 传递不同的参数来依次调用预处理器、编译器、汇编器和连接器。当然实际使用 ucc 命令时，我们不会绕这么大一个圈子，直接用“ucc hello.c -o hello”即可得到可执行程序 hello。

```
iron@ubuntu:demo$ ucc -E hello.c -o hello.i
iron@ubuntu:demo$ ucc --dump-ast --dump-IR -S hello.i -o hello.s
iron@ubuntu:demo$ ucc -c hello.s -o hello.o
iron@ubuntu:demo$ ucc hello.o -o hello
```

在这一小节，我们进入目录 ucc\driver 来分析一下 ucc 驱动器的源代码。图 1.22 给出了 ucc\driver\linux.c 中的部分代码，第 7 至 12 行是我们使用“ucc -E hello.c -o hello.i”命令时，ucc 驱动真正执行的命令，其中的 CPP 代表的是 C 预处理器（C PreProcessor），而非 C Plus Plus。通过“-D”参数指定了第 8 行的\_\_STRICT\_ANSI\_\_ 等预定义的宏，而第 11 行的\$1、\$2 和 \$3 充当占位符的作用，实际使用时会被 C 程序员所指定的参数所替代。例如当 C 程序员输入以下命令时：

```
ucc -E -v hello.c -I../ -DREAL=double -o hello.ii
```

命令中的“-I../ -DREAL=double”会被 UCC 驱动提取出来，用于替换图 1.22 第 11 行的“\$1”；而命令中的“hello.c”用于替换第 11 行的“\$2”，充当预处理器的输入文件；命令中的“hello.ii”则替换“\$3”，用来存放预处理后的结果。而“-v”参数会使 UCC 驱动在终端上打印出实际所用到的命令，如下所示：

```
/usr/bin/gcc -U_GNUC_ -D_POSIX_SOURCE -D_STRICT_ANSI_ -Dunix -Di386
-Dlinux -D_unix_ -D_i386_ -D_linux_ -D_signed_=signed -D_UCC
-I/home/iron/bin/include -I../ -DREAL=double -E hello.c -o hello.ii
```

图 1.22 第 21 至 23 行是 C 编译器（C Compiler）对应的命令，当我们输入以下命令时

```
ucc -S -v hello.c --dump-ast -o hello.asm
```

实际调用的是如下命令，图 1.22 第 22 行的\$1、\$2 和 \$3 的含义跟第 11 行类似。而 ucl 就是我们后续要剖析的 UCC 编译器，相关源代码在目录 ucc\ucl 中。

```
/home/iron/bin/ucl -o hello.asm --dump-ast hello.i
```

图 1.22 第 24 至 30 行则对应的是汇编器（ASsembler）和连接器（Linker）。第 29 行的参数“-lc”和“-lm”用来通知连接器，我们需要 C 函数库 libc.so 和数学运算函数库 libm.so。

```
1 /*****
2 ucc -E -v hello.c -I../ -DREAL=double -o hello.ii
3   $1      -I../ -DREAL=double,    command-line options
4   $2      hello.c,           input file
5   $3      hello.ii,          output file
6 *****/
7 char *CPPProg[] = {
8   "/usr/bin/gcc", "-U_GNUC_", "-D_POSIX_SOURCE", "-D_STRICT_ANSI_",
9   "-Dunix", "-Di386", "-Dlinux", "-D_unix_", "-D_i386_",
10  "-D_linux_",     "-D_signed_=signed", "-D_UCC",
```

```

11   "-I" UCCDIR "include", "$1", "-E", "$2", "-o", "$3", 0
12 };
13 ****
14 ucc -S -v hello.c --dump-ast -o hello.asm
15 that is,
16 ucl -o hello.asm --dump-ast hello.i
17 $1 --dump-ast, some command-line options
18 $2 hello.i,    input file
19 $3 hello.asm,  output file
20 ****
21 char *CCProg[] = {
22   UCCDIR "ucl", "-o", "$3", "$1", "$2", 0
23 };
24 char *ASProg[] = {
25   "/usr/bin/as", "-o", "$3", "$1", "$2", 0
26 };
27 char *LDProg[] = {
28   "/usr/bin/gcc", "-o", "$3", "$1", "$2",
29   UCCDIR "assert.o", "-lc", "-lm", 0
30 };

```

图 1.22 编译命令

接下来的问题就是如何在 ucc 驱动中开启一个新进程来执行上述命令行所对应的预处理器、编译器、汇编器或者连接器。图 1.23 给出了在 Linux 平台上创建子进程及装载执行新程序的过程，其中关键的函数有 fork()、execv() 和 wait() 这三个系统调用。Linux 源于 Unix，其系统调用 fork() 也源于 Unix。生物界中，有许多细胞进行繁殖时，是通过细胞一分为二的分裂来进行，Linux 创建子进程的系统调用 fork() 也相当于这个过程。英文 fork 是叉子的意思，叉子一分为二的形状也很形象地描绘了这个细胞分裂的过程。这实际上是一种克隆（clone）行为。除了进程号 PID、父进程号 PPID 及 fork() 返回值等少量信息不一样外，新创建的子进程和父进程几乎一模一样。这意味着当 fork() 成功后，会有两个进程并发执行以下第 4 行开始的代码，当然对这两个进程而言，其 fork() 函数的返回值是不一样的，新创建的子进程会执行第 8 至 11 行的代码，而父进程则执行第 17 行之后的代码。如果希望子进程做点和父进程不一样的工作，则需要在子进程中加载外存中的可执行程序，这可借助系统调用 execv() 来实现。我们知道，编译链接后生成的可执行程序一般是存放在外存的，只有被装载器（Loader）从外存载入内存后，这个可执行程序才能运行。而 Linux 的 execv() 系统调用就起到了装载器的作用。子进程通过 execv() 成功加载一个可执行程序后，其整个进程空间中的代码段和数据段就会被替换成新加载的可执行程序的代码段和数据段。这意味着一旦图 1.23 第 8 行的 execv() 成功执行，子进程就不再执行第 9 行的代码了。当然 execv() 不一定成功，其原因可能是程序员在第 8 行提供的参数 cmd 中的路径不正确，或者该路径对应的文件根本就不是一个可执行程序，这时会执行第 9 至 11 行的代码进行错误处理。父进程通过执行第 17 行中的系统调用 wait() 来等待子进程结束，这相当于父子进程之间的简单同步。只有在子进程执行完或者出错时，父进程才会执行第 19 至 25 行之间的代码。UCC 驱动会通过函数 ParseCmdLine() 分析程序员提供的命令行参数，并结合图 1.22 中的 CPPProg、CCProg、ASProg 和 LDProg 等命令模板，通过函数 BuildCommand() 来构建实际要调用的命令，并将其作为参数传给图 1.23 中的 Execute() 函数。函数 ParseCmdLine() 和 BuildCommand() 的代码在 ucc\driver\ucc.c 中，主要是进行一些字符串的处理，这里不再啰嗦。

```

1 int Execute(char **cmd) {
2   int pid, n, status;
3   pid = fork();

```

```

4   if (pid == -1){
5       fprintf(stderr, "no more processes\n");
6       return 100;
7   }else if (pid == 0){
8       execv(cmd[0], cmd);
9       perror(cmd[0]);
10      fflush(stdout);
11      exit(100);
12  }
13 /*****
14 wait(): on success, returns the process ID of the
15 terminated child; on error, -1 is returned.
16 ****/
17 while ((n = wait(&status)) != pid && n != -1){
18 }
19 if (n == -1){
20     status = -1;
21 }
22 if (status & 0xff){
23     fprintf(stderr, "fatal error in %s\n", cmd[0]);
24     status |= 0x100;
25 }
26
27 return (status >> 8) & 0xff;
28 }

```

图 1.23 Execute()

## 1.5 结合 C 语言来学汇编

### 1.5.1 汇编语言简介

汇编语言的语法和语义都不复杂，如果会 C 语言，那就一定能在很短时间内看懂本节介绍的汇编语言，但是要很熟练地阅读和编写汇编程序，则需要长期的训练。上世纪 80 年代末，求伯君闭关一年多，单枪匹马用洋洋洒洒数万行汇编代码开发出 DOS 版 WPS，由此奠定江湖大佬地位，但时至今日，这样的个人英雄主义时代已经一去不复返。除了信息安全和软件逆向分析等少数领域外，需要使用汇编语言进行研发的场合已经不多了，用高级语言进行开发已经是主旋律，即便是 Linux 内核和嵌入式驱动开发，也几乎是 C 语言的天下。不论杰出的程序员能用汇编语言写出多么精巧的代码，在市场经济时代，花最少的钱获得最大的回报，永远是企业所追逐的。随着编译优化技术的发展，由编译器自动生成的汇编代码的执行效率，已经不输于大部分汇编程序员手工编写的汇编代码。而汇编代码的可读性差，开发效率相对较低，由此导致的开发和维护成本高，注定了高级语言取而代之成为编程的主流。

CPU 的指令集实际上是软件和硬件的接口，而汇编语言就以助记符的形式描述了这个指令集。严格说来，CPU 并不认识汇编语言，CPU 只认识与这些助记符对应的 0/1 机器码，但习惯上把汇编语言和机器语言统称为低级语言。学习汇编语言可以熟悉软件与硬件的接口，毕竟指令集是计算机体系结构的一部分。当然，我们要剖析的是 C 编译器，而 C 编译器最终需要把 C 源代码翻译成汇编代码，在开始 C 编译器剖析的万里长征时，有必要熟悉一下我们最终要到达的目的地。学习任何语言的不二法宝是“先模仿然后主动地去使用这门

语言来表达”，学英语、学 C 语言和学汇编都是如此。为了能用尽量少的篇幅介绍 UCC 编译器用到的汇编语言，我们采用的是结合 C 语言来熟悉汇编的办法。我们先简介一些基本概念，然后给出一个简单的 C 程序及由 UCC 编译器生成的汇编代码。通过 C 源代码，我们可以知道程序要完成的功能，然后去阅读汇编代码，就会容易许多。此处，我们要达到的目标是“能基本看懂 UCC 编译器用到的汇编代码”。确切地说，是能基本看懂 Linux 平台对应的 `ucc\ucl\x86Linux.tpl` 中的汇编代码，而 Windows 平台对应的 `ucc\ucl\x86win32.tpl` 只在语法格式上有所不同，两者没有本质区别。文件名中 `x86Linux.tpl` 的“`.tpl`”是模板（template）的缩写，这意味着其中的汇编代码是不完整的，需要指定具体的操作数后才能构成完整的汇编指令。例如，在以下所示的汇编模板中，`%0`、`%1` 和 `%2` 是占位符，这和前一节图 1.22 中的`$1`、`$2` 和 `$3` 的作用类似。以下两条汇编指令用于比较`%2` 和 `%1` 对应的两个有符号整数，如果两者不相等，则跳转到`%0` 对应的目标地址处；若相等，则执行下一条汇编指令。助记符 `cmp` 是 `compare` 的缩写，`cmpl` 中的后缀“1”表示进行 32 位的比较；而 `jne` 是 `Jump if Not Equal` 的缩写。

```
TEMPLATE(X86_JNEI4,      "cmpl %2, %1; jne %0")
```

出于系统安全方面的考虑，就如网站的管理员和一般用户具有不同的权限一样，CPU 的指令集也被划分为特权指令和非特权指令。CPU 处于可以使用任何指令的状态时，被称之为“管态或核心态”，管态是处于核心的管理状态的意思；而 CPU 处于只能使用非特权指令时，被称为“目态或用户态”，目态是表面上看到的外层的非核心的状态。操作系统内核要管理所有的软硬件资源，就要有权使用 CPU 的所有指令，要处于管态。而操作系统内核之外的应用程序只使用非特权指令，处于目态。当然，x86 CPU 有四个不同的运行级别，从 `ring0` 一直到 `ring3`，但 Linux 实际用到的只有 `ring0` 和 `ring3`。因此，我们仍可用管态和目态的概念来理解 x86 CPU 所处的运行级别。像开中断 `sti`、关中断 `cli`、输入 `in` 和输出 `out` 这样的指令在 x86 中都是特权指令，只有 OS 内核才能使用。如果一个应用程序要读写文件，而读写文件显然需要做输入输出，这就得向操作系统内核提出申请。真正进行文件读写的权限是在操作系统内核中，应用程序通过触发中断来申请操作系统内核提供的各种服务，这就是所谓的系统调用。

此处，我们可能会有这样的疑问，“为什么需要绕这么个圈子，为什么不让应用程序有权使用所有指令，这样应用程序就不用麻烦操作系统内核了，且消除了管态和目态的切换开销，运行效率还可以大大提高”。答案就是前文中提及的“系统安全”，可以想象如果网站的一般用户和管理员拥有一样的权力，那带来的只能是混乱。在标准 C 语言里，并没有任何语法结构与开关中断这样的汇编指令相对应，而我们知道，Linux 内核主要是用 C 语言来开发的。这就意味着，我们需要直接用汇编语言来编写这部分代码，或者由 C 编译器提供在 C 语言中嵌入汇编代码的机制。而我们要剖析的 UCC 编译器暂时还未提供在 C 代码中嵌入汇编的机制。

在应用程序中，我们可以通过“`int` 或 `sysenter`”这样的汇编指令来触发一个中断，从而来向内核发出系统调用请求。例如在 Linux 平台中，C 程序员可调用 `read()` 和 `write()` 这样的库函数来向内核发出读文件和写文件的系统调用请求，但是这些库函数最终还是需要去调用汇编指令“`int` 或 `sysenter`”，只不过这些工作由库函数的实现者帮我们代劳了而已。

站在 C 编译器的角度来看，我们需要从 CPU 的指令集中选取一些汇编指令来支持 C 语言的语义，`ucc\ucl\x86Linux.tpl` 中的汇编代码就是 UCC 编译器的选择。这个选择并非最优，也不唯一，但非常清晰易懂。这些汇编代码只是 x86 指令集中的一部分，但已足够支撑起 C 语言的语义。无论多复杂的 C 语言代码，只要是用 UCC 编译器来编译，我们最终得到的汇编程序都是由 `ucc\ucl\x86Linux.tpl` 中的汇编指令组合而来。

页式存储管理是许多现代 CPU 普遍支持的内存管理机制，而 Intel 的 x86 CPU 要兼容早

期 8086 芯片的分段机制，最终形成的是“先分段再分页”的段页式存储管理。如何利用或者干脆绕开 x86 的分段机制，及如何开启请求分页机制的工作已经由操作系统内核完成，操作系统可在 x86 CPU 上，为每个运行的程序（进程）营造一片相对独立的一维地址空间。

我们知道，平面是个二维空间，要定位二维空间中的一个位置我们需要坐标(x,y)；而要定位一条线段中的某个点，我们只需要一个坐标 x，这个坐标 x 也就是平时我们所说的地址。这里，我们不去关注操作系统如何为每个进程构建一片独立的一维虚地址空间，我们更关注对每个进程而言，这片地址空间是如何组织的。在我们写的 C 程序里有代码，也有数据，因此，进程的地址空间逻辑上可被看成由“代码区”和“数据区”这两部分构成。全局变量、静态变量和常量在编译时，就可知它们要占多大空间，这部分空间被称为静态数据区。静态数据区中“静态”的含义是指这部分数据区在“编译时”可确定，不需要等到“程序运行时”才进行分配。对应的，就有“动态数据区”。在函数调用时，我们需要借助一个“栈”来存放函数对应的局部变量、临时变量和返回地址等信息，而如果一个 C 语言函数从未被调用，我们就不需要在栈中为该函数分配空间，这意味着 C 函数中的局部变量是在栈中动态分配的。

这块动态数据区被称为栈（stack）的原因在于，函数的调用正好满足先进后出的关系，即最先被调用的函数却最后一个返回。另一个动态分配的例子是 C 语言中的 malloc() 函数，或 C++ 中的 new 操作，这部分动态数据区被称为堆（Heap）。我们平时会讲“这一堆乱七八糟的东东”，“堆”反映的是无序。通过 malloc 或者 new 分配得到的数据区，其回收操作是无序的，在 C 和 C++ 中需要由程序员手工完成。所谓的“内存泄露”往往指的也是 C 程序员调用了 malloc，却忘了 free；或者 C++ 程序员调用了 new，却忘了对应的 delete。既然手工来管理堆空间要看程序员的人品，于是 Java 等语言就引入了垃圾自动回收机制，减轻了程序员的负担，当然这不代表 Java 程序中就不存在内存泄露的问题。总之，我们可把一个进程的地址空间分为以下几部分：

- (1) 代码区
- (2) 数据区      动态数据区（由堆和栈构成，运行时分配）
  - 静态数据区（全局变量、静态变量和常量，编译时确定）

需要由 C 编译器管理的数据区主要是栈，而堆空间的管理则通过相应的 C 库函数 malloc 和 free 来实现，由库函数实现者去花心思，不用劳烦 C 编译器。C 编译器对栈的管理，主要是通过生成函数调用和函数返回的相应汇编指令来实现，这些指令在运行时，就会自动实现“栈”空间的分配和回收，因此，栈空间的管理不需要 C 程序员操心。需要说明的是，数据区一般是用来存放数据，但是黑客也可以利用缓冲区溢出等漏洞，在栈或堆中注入一些人为构建的机器代码。

下面我们以图 1.24 中的 C 语言代码来举例说明动态和静态数据区的概念。按 C 语言的语法，图 1.24 中的 str1 是全局变量，而 str2 是静态变量，而第 12 行的"Hello World"则是常量，这部分数据就保存在我们前文中所述的“静态数据区”；而函数 h 中的变量 str3 是个局部变量，在函数 h 被调用时，才会在栈中动态为 str3 分配内存，在第 9 行，我们让 str3 指向了由 malloc 分配的 16 字节的堆空间。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 char * str1;
4 void h(char * str){
5   static char * str2;
6   char * str3;
7
8   str2 = str;
9   str3 = (char*)malloc(16);

```

```

10 }
11 int main(int argc,char * argv[]){
12   str1 = "Hello World";
13   h(str1);
14   return 0;
15 }
```

图 1.24 用于说明数据区的 C 程序

图 1.25 是由“ucc -S hello.c”命令生成的部分汇编代码。图 1.25 第 8 行的 str1 对应上述 C 代码中的 str1，而第 9 行的 str2.0 则对应上述 C 代码中的静态变量 str2。为了避免与全局变量重名，UCC 为静态变量 str2 加上了“.0”的后缀。图 1.25 第 8 行中的“.comm str1,4”告诉汇编器，str1 是个不带初值的全局变量，要占 4 个字节，在 32 位机器中，指针一般就占 4 个字节；而第 9 行的“.lcomm str2.0,4”则表示 str2.0 是个只在当前文件中使用的不带初值的静态变量，要占 4 个字节。按 C 的语义，不带初值的全局变量和静态变量会被初始为 0，这个工作一般由装载器来实现。

图 1.25 第 5 行对应 C 代码中的字符串常量"Hello World"，UCC 编译器将其命名为“.str0”，由于合法的 C 语言中不存在以“.”为前缀的变量名，我们可以不用担心“.str0”会跟 C 代码中的变量重名，而第 5 行的“.string”则指明了紧随其后的是以`\0'结束的字符串"Hello World"。UCC 编译器对静态数据区的管理只要做到这一层即可，为这些变量分配具体存储位置的工作就由后续的汇编器和连接器来完成。而图 1.25 中的第 3 行“.data”则告诉汇编器接下来的是静态数据区，而第 11 行的“.text”则告诉汇编器接下来的又切换成代码区了。在图 1.25 中我们没有发现形参 str 和局部变量 str3，不过，对比图 1.24 的 C 代码，应可以猜出图 1.25 第 15 至 35 行就是 C 函数 h() 所对应的汇编代码，在后面的分析中，我们会看到图 1.25 第 23 行的 20(%ebp) 即为形参 str，而第 28 行的-4(%ebp)是局部变量 str3。

```

1 # Code auto-generated by UCC
2
3 .data
4
5 .str0:    .string "Hello World"
6
7
8 .commstr1,4
9 .lcomm    str2.0,4
10
11 .text
12
13 .globl    h
14
15 h:
16   pushl %ebp
17   pushl %ebx
18   pushl %esi
19   pushl %edi
20   movl %esp, %ebp
21   subl $4, %esp
22 .BB0:
23   movl 20(%ebp), %eax
24   movl %eax, str2.0
25   pushl $16
26   call malloc
27   addl $4, %esp
28   movl %eax, -4(%ebp)
29   movl %ebp, %esp
30   popl %edi
31   popl %esi
32   popl %ebx
33   popl %ebp
34   ret
35
```

图 1.25 静态数据区及函数 h() 对应的汇编代码

而要比较好的理解函数 h 对应的汇编代码，我们需要再介绍一下寄存器的概念。在物理上，X86 汇编代码中描述的寄存器位于 CPU 中，而静态数据区则存在于内存中。由于寄存器的访问速度远远快于内存，出于速度的考虑，我们当然期望汇编指令的操作数能尽量地使用寄存器。直观上，CPU 提供的寄存器数量似乎越多越好，但这有硬件成本的约束，而且计算机研究人员还发现当寄存器数量到一定程度时，再增加寄存器并没有带来性能上的明显改进。性能的改进在软件上相当程度上取决于编译器对寄存器的分配和指派算法，还有优化，

在硬件上更多的着眼点是放在 CPU 和内存之间的 Cache 上。UCC 编译器使用的 x86 寄存器有 eax、ebx、ecx、edx、esp、ebp、esi 和 edi 等 8 个寄存器。以 e 为前缀命名的寄存器是 32 位的，若去掉前缀 e 得到的 ax、bx、cx、dx、sp、bp、si 和 di 等寄存器就是 16 位的。寄存器 ax 还可分为 ah 和 al 这两个 8 位的寄存器，其中的 h 表示高字节，而 l 表示低字节，bx、cx 和 dx 寄存器也存在类似的 8 位寄存器。Intel 公司早期的 8086 是 16 位的 CPU。“如何给变量和函数取名”估计是一直困扰程序员的难题。Intel 后来在 64 位 CPU 中新加的寄存器干脆就叫 R8、R9、... 和 R15。不过，对这些通用的寄存器，确实也想不出什么更好的名字。图 1.24 中的 main() 函数需要使用寄存器，而被调函数 h() 也需要使用寄存器，因此，要在主调函数（caller）和被调函数（callee）之间定个规矩，哪些寄存器要由主调函数进行保存，哪些寄存器要由被调函数进行保存。

IT 大佬们开讨论后的结果是：eax、ecx 和 edx 被划入易失寄存器（scratch register），需要由主调函数来保存，此处“保存”一般指的是把寄存器的数据写回内存。当然，如果主调函数在调用“被调函数”时，压根儿就没使用易失寄存器，或者易失寄存器中的数据不需要保存（比如这些数据从内存载入寄存器后，从来没有被修改过），那就没必要做保存动作了，毕竟，写内存有一定的时间开销。而 ebx、esi 和 edi 被划入保值寄存器（preserve registers），需要由被调函数来保存，这意味着如果主调函数在保值寄存器中存放了有用的数据，在函数调用返回后，保值寄存器中的数据仍然要和函数调用前是一样的，这也是“保值”一词的由来。但是，被调函数发现寄存器不够用时，也可先保存一下“保值寄存器”的值，再借用保值寄存器，使用后再恢复成原样即可。UCC 编译器使用了最简单的策略来管理“保值寄存器”，不论本函数是否需要使用这些寄存器，在函数入口处都把保值寄存器先压入到栈中，函数返回时，再从栈中逆序弹出，即图 1.25 中第 17 至 19 行看到的三条 push 指令，及第 30 至 32 行看到的三条 pop 指令。当然，这不是最优的策略，但应是最简单最好理解的。而每个函数都可能成为被调函数，在由 UCC 编译器产生的汇编代码中，每个函数的开头部分（prologue）和结束部分（epilogue）的汇编代码都是一样的，如下所示：

```
TEMPLATE(X86_PROLOGUE,
    "pushl %%ebp;pushl %%ebx;pushl %%esi;pushl %%edi;movl %%esp, %%ebp")
TEMPLATE(X86_EPILOGUE,
    "movl %%ebp, %%esp;popl %%edi;popl %%esi;popl %%ebx;popl %%ebp;ret")
```

当一个 C 函数被调用时，我们需要在栈中为该函数分配一段内存空间，我们需要记录这段内存的两端，x86 提供的寄存器 ebp 和 esp 正是用于此目的。x86 的入栈方向一般是从高地址往低地址方向入栈，如图 1.26 所示。

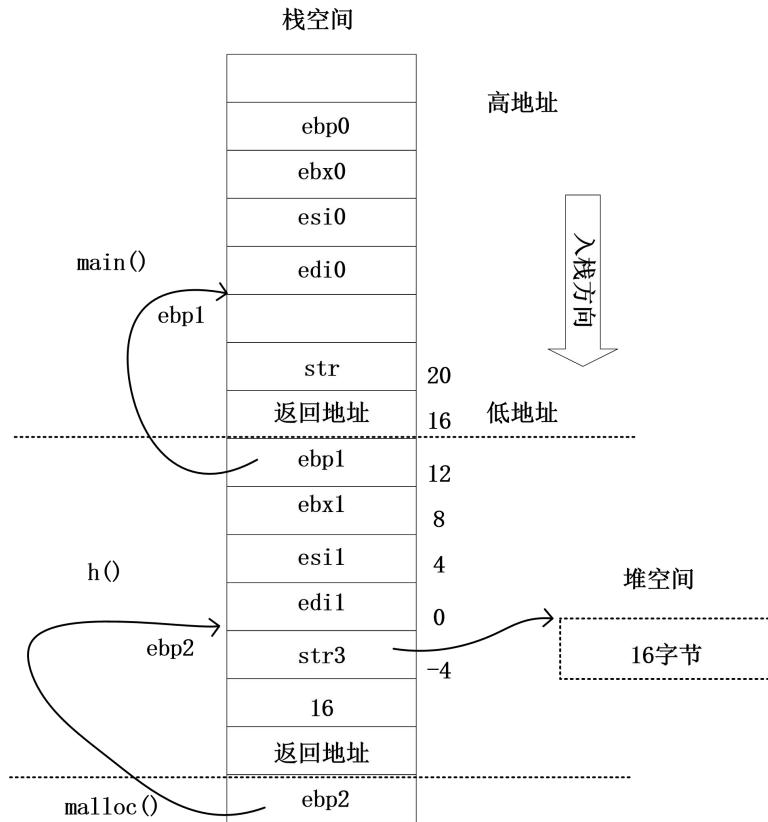


图 1.26 栈示意图

我们再来看一下 main() 函数对应的汇编代码，如图 1.27 所示。图 1.27 中第 39 至 43 行的四条 push 指令把 ebp、ebx、esi 和 edi 入栈，对应图 1.26 中的 ebp0、ebx0、esi0 和 edi0。第 43 行的 movl 指令可用于“内存与寄存器”或“寄存器与寄存器”之间的数据传递，mov 是 move 的缩写，后缀 l 表示进行 32 位的操作，后面我们还会看到表示 16 位操作的后缀 w 和表示 8 位操作的后缀 b，分别是 long, word 和 byte 的缩写。严格说来，32 位 CPU 的字长是 32 位，但由于 word 这个词语已在 16 位时代广泛使用，因此很多时候我们还是把一个字当作 2 个字节。图 1.27 第 43 行用于把寄存器 esp 中的数据传送到寄存器 ebp 中，相当于我们为 main 函数在栈中的数据确立了基址，要查找栈中的其他数据，都会以 ebp 为基准点，例如前文中所述的 20(%ebp) 和 -4(%ebp)，寄存器 ebp 是 Base Pointer 的缩写。而 esp 则始终指向栈顶，是 Stack Pointer 的缩写。第 44 行的 sub 指令是 subtract 的缩写，相当于作 “esp -= 4;” 的计算，这形成了图 1.26 中 ebp1 所指向的位置下方的一处空白，这位置对应一个临时变量，用来存放字符串“Hello World”的地址。UCC 编译器经过优化后，直接在第 46 行把字符串“Hello World”的地址存到寄存器 eax 中了。第 46 行中的 lea 指令是 Load Effective Address 的缩写，我们知道一个内存单元相当于一个房间，其地址相当于房间门牌号，该内存单元的数据相当于房间中的物品，此处 lea 指令用于取.str0 对应静态数据区的首地址，并存放到寄存器 eax 中。图 1.27 第 47 行的 mov 指令把寄存器 eax 的数据传送到全局变量 str1 中，这正是图 1.24 中第 12 行 “str1 = "Hello World";” 所要完成的功能。图 1.27 第 48 至 50 行则对应的是 C 函数调用 “h(str1);”，第 48 行的 push 指令完成了实参 str1 给形参 str 赋值的操作，图 1.26 中的 str 标记了函数 h 形参的位置。第 49 行的 call 指令会先把返回地址压入栈中，这样当函数调用 h(str1) 返回后，我们才能回到图 1.27 的第 50 行处。图 1.27 第 51 行用于把返回值存放到寄存器 eax 中。

<pre> 36 .globl  main 37 38 main: </pre>	<pre> 39  pushl %ebp 40  pushl %ebx 41  pushl %esi </pre>
--	---

```

42 pushl %edi
43 movl %esp, %ebp
44 subl $4, %esp
45 .BB1:
46 leal .str0, %eax
47 movl %eax, str1
48 pushl str1
49 call h
50 addl $4, %esp
51 movl $0, %eax
52 movl %ebp, %esp
53 popl %edi
54 popl %esi
55 popl %ebx
56 popl %ebp
57 ret
58

```

图 1.27 main() 函数对应的汇编代码

IT 厂商们需要对“函数参数如何传递、返回值如何存放、及函数调用结束后由主调函数还是被调函数进行参数退栈”等操作进行约定，这被称为调用约定（Calling Convention）。有许多不同的函数调用约定，UCC 编译器使用的是 C 调用约定，即按从右到左的顺序把参数入栈，由主调函数负责参数退栈，图 1.27 第 50 行对 esp 进行加 4 的操作就是用于把函数 h 的参数出栈。C 的调用约定使得 C 语言可以支持变参函数。当 C 编译器编译变参函数时，并不知道该函数被调用时会有多少个实参入栈，这个信息只有主调函数知道，因此要由主调函数负责退栈。由图 1.26 所示的 UCC 编译器栈内存布局，可以发现，对被调函数来说，最后入栈的参数所处的栈内存位置始终都是 ebp+20，若按从右到左的顺序把参数入栈，则以下变参函数 printf 的第一个参数 format 的位置就是 ebp+20；如果按从左到右的顺序入栈，第一个参数 format 会先被入栈，由于参数个数可变，format 的位置对被调函数 printf 来说是不可知的。

```
int printf(const char *format, ...);
```

有了这样的基础后，我们再回头去看图 1.25 中的 h() 函数对应的汇编代码，当 h() 函数成为当前正在执行的函数时，寄存器 ebp 会被设置到图 1.26 的 ebp2 处，而 ebp+20 则正好是形参 str 的地址，图 1.25 第 23 行的指令“movl 20(%ebp), %eax”会把 ebp+20 处的内容传递到寄存器 eax 中。图 1.25 第 25 至 27 行实现了函数调用 malloc(16)，按照调用约定，malloc 的返回值会保存在寄存器 eax 中，图 1.25 第 28 行将寄存器 eax 的内容保存到(ebp-4)处，即把 malloc() 的返回值存到局部变量 str3 中。图 1.25 第 33 行弹出在栈中存放的 ebp1 并存到寄存器 ebp 中，这会使 ebp 再次指到图 1.26 main() 函数对应的 ebp1 处。图 1.25 第 34 行的 ret 指令则从当前栈顶弹出返回地址并存放到程序计数器 eip 中，对 x86 CPU 而言，寄存器 eip 中存放了下一条要执行指令的地址，从 h() 函数返回后，就会执行图 1.27 的第 50 行。由于图 1.25 第 33 行已经把 ebp 指针恢复到 ebp1 处，从 h() 函数返回 main() 函数后，通过 ebp 得到的局部变量就是 main() 函数中的局部变量了。

纸面上的表达很难达到动态的效果，更好的办法是对照着图 1.24、图 1.25 和图 1.27，然后拿起笔，在纸上画出每一条汇编指令执行完后的栈示意图，如图 1.26 所示。需要注意的是 esp 始终指向顶栈，由于入栈方向是由高到低，因此 pushl 指令会使栈顶指针 esp 减 4，popl 则使栈顶指针 esp 加 4。而 call 指令会将其下一条指令的地址（即函数返回地址）入栈，这相当于作了 esp 减 4 的操作，而 ret 指令相当于做了 esp 加 4 的出栈操作。

在下一节，我们将再写几个简单的 C 程序，来进一步解释 ucc\ucl\x86Linux.tpl 中的其他汇编代码。

## 1.5.2 整数运算

让我们还是从熟悉的加减乘除等算术运算入手。图 1.28 给出了一个简单的 C 程序，其中包含了常见的 C 语言算术运算，我们依旧采用对比 C 语言代码和 UCC 编译器生成的汇编代码的方法来讨论。

```

1 int a = 10, b = 20, c;
2 int main(int argc, char * argv[]){
3     static int d ;
4     static int e = 5;
5     c = a | b;
6     c = a & b;
7     c = a << 2;
8     c = a >> 2;
9     c = a + b;
10    c = a - b;
11    c = a * b;
12    c = a / b;
13    c++;
14    c--;
15    c = a % b;
16    c = -a;
17    c = ~a;
18    return 0;
19 }
20

```

图 1.28 arith.c

我们有意在图 1.28 的第 1 至 4 行定义了几个初始化或未初始化的变量。图 1.29 给出了这几个变量在汇编代码中的区别。在图 1.29 中，第 7 行至第 9 行对应的是初值为 10 的全局变量 a；而第 11 至 13 行对应的是初值为 20 的全局变量 b；第 15 行对应的是不带初值的全局变量 c；而第 16 行对应的是不带初值的静态变量 d，为避免重名，被 UCC 编译器改名为“d.0”；而第 17 行对应的是初值为 5 的静态变量 e，被 UCC 编译器重命名为“e.1”。在汇编代码中不存在“.global e.1”，而在第 7 行有“.globl a”，在第 11 行有“.globl b”，这代表变量名 a 和 b 在全局范围可见，但是 e.1 只在当前文件中可见。按照 C 语言的语义，不带初值的全局或静态变量其缺省值一般为 0。在生成目标文件(后缀为.obj 或.o 的文件)时，我们只需要在目标模块中记录这块要被初始化为 0 的空间有多大就可以，没有必要把一堆的 0 存到目标文件中。

```

1 # Code auto-generated by UCC
2
3 .data
4
5
6
7 .globl a
8
9 a: .long 10
10
11 .globl b
12
13 b: .long 20
14
15 .comm c,4
16 .lcomm d.0,4
17 e.1: .long 5
18
19
20 .text
21
22 .globl main
23
24 main:
25 pushl %ebp
26 pushl %ebx
27 pushl %esi
28 pushl %edi
29 movl %esp, %ebp
30 subl $48, %esp
31 .BB0:

```

图 1.29 初始化和未初始化的变量

让我们看一下图 1.28 第 5 至 17 行各算术运算所对应的汇编代码，如图 1.30 所示。图 1.30 第 32 至 33 行对应“c = a | b;”，第 32 行把 a 的值从内存加载到寄存器 eax 中，第 33 行把内存中的变量 b 和寄存器 eax 作“按位或运算”，结果仍存于 eax 中，第 34 行再把运算结果从寄存器 eax 写回到变量 c。图 1.30 第 35 至 49 行所做的运算依次为“按位与运算 andl”、“左移运算 shll”、“算术右移运算 sarl”、“加法运算 addl”和“减法运算 subl”。

```

32 movl a, %eax
33 orl b, %eax
34 movl %eax, c
35 movl a, %ecx
36 andl b, %ecx
37 movl %ecx, c
38 movl a, %edx
39 shll $2, %edx
40 movl %edx, c
41 movl a, %ebx
42 sarl $2, %ebx
43 movl %ebx, c
44 movl a, %esi
45 addl b, %esi

```

```

46    movl %esi, c
47    movl a, %edi
48    subl b, %edi
49    movl %edi, c
50    movl a, %eax
51    imull b, %eax
52    movl %eax, c
53    movl a, %eax
54    cdq
55    idivl b
56    movl %eax, c
57    incl c
58    movl c, %edx
59    addl $-1, %edx
60    movl %edx, c
61    movl a, %eax
62    cdq
63    idivl b
64    movl %edx, c
65    movl a, %eax
66    negl %eax
67    movl %eax, c
68    movl a, %eax
69    notl %eax
70    movl %eax, c
71    movl $0, %eax
72    movl %ebp, %esp
73    popl %edi
74    popl %esi
75    popl %ebx
76    popl %ebp
77    ret
78

```

图 1.30 arith.s

右移运算比较特殊，有“算术右移 sar”和“逻辑右移 shr”之分，sar 是 Shift Arithmetic Right 的缩写，而 shr 则是 Shift Right 的缩写。进行算术右移时最高位要用之前的符号位来填充，而逻辑右移时最高位补 0。例如，以下函数 f() 会陷入死循环，而函数 g() 则不会。如下代码中的 a 是有符号整数，C 编译器会选用算术右移指令 sar 来进行移位操作，而 a 的值为 -1，在内存中以补码形式存放时为 0xFFFFFFFF，算术右移后仍然是 -1。而 b 为无符号整数，C 编译器会选择逻辑右移指令 shr 来进行移位，最高位补入的是 0，经过 32 次的逻辑右移操作后，b 的值就变为 0 了。

```

int a = -1;
unsigned int b = 0xFFFFFFFF;
void f(){
    while(a >= 1);
}
void g(){
    while(b >= 1);
}

```

图 1.30 中的第 50 至 52 对应的是“ $c = a * b;$ ”，其中第 51 行的 imul 是有符号整数乘法指令，而无符号整数乘法指令为 mul。第 53 至第 56 行对应的是“ $c = a / b;$ ”，第 53 行把被除数 a 存到寄存器 eax 中，第 54 行的指令 cdq 是 Change Double word to Quadrate word 的缩写，即把双字扩展为四字。在 16 位 CPU 时代，把一个 word 定义为 2 字节，双字对应 4 字节，四字对应 8 字节。x86 会把被除数存于寄存器 edx 和 eax 中，edx 充当高 32 位，而 eax 充当低 32 位。若寄存器 eax 最高位为 1，执行 cdq 指令之后，寄存器 edx 里的各个位都是 1；否则全为 0。而如果要进行无符号整数的除法，由于 eax 的最高位不再被当作符号位，我们直接把 edx 寄存值赋值为 0 即可，uclx86Linux.tpl 中与此有关的代码如下所示：

```
TEMPLATE(X86_DIVU4,      "movl $0, %%edx;divl %2")
```

图 1.30 第 55 的指令 idiv 是有符号整数除法指令，而 div 是无符号整数除法指令。整数除法运算的商会被存放在寄存器 eax 中，如图 1.30 第 56 行所示；而余数则被存放在寄存器 edx 中。当我们进行“ $c = a \% b;$ ”的取余运算时，就需要用到除法运算的余数，如图 1.30 第 61 至 64 行所示。图 1.30 第 57 行的 inc 指令是 increment 的缩写，用于对操作数进行加 1 操作。第 65 至 67 行对应的是“ $c = -a;$ ”，其中第 66 行的指令 neg 是 negative 的缩写，用于取“相反数”。如果有符号数 a 为 -1，即 0xFFFFFFFF，经过 neg 指令的处理后，就得到 a 的相反数 1，即 0x00000001。而第 68 至 70 行对应的是“ $c = \sim a;$ ”，第 69 行的 not 指令用于按位

取位，如果 a 为-1，则按位取反为得到的是 0x00000000。

接下来，让我们来讨论一下 UCC 编译器中用到的有条件跳转指令。图 1.31 给出了一个简单的 C 程序，第 14 至 44 行是 main 函数对应的汇编代码。

```

1 int a = 10,b = 20,c;
2 int main(int argc,char * argv[]){
3     if(!a){
4         c++;
5     }
6     if(a == b){
7         c++;
8     }
9     if(a > b){
10        c++;
11    }
12    return 0;
13 }
14 main:
15 pushl %ebp
16 pushl %ebx
17 pushl %esi
18 pushl %edi
19 movl %esp, %ebp
20 .BB0:
21 cmpl $0, a
22 jne .BB2
23 .BB1:
24 incl c
25 .BB2:
26 movl a, %eax
27 cmpl b, %eax
28 jne .BB4
29 .BB3:
30 incl c
31 .BB4:
32 movl a, %eax
33 cmpl b, %eax
34 jle .BB6
35 .BB5:
36 incl c
37 .BB6:
38 movl $0, %eax
39 movl %ebp, %esp
40 popl %edi
41 popl %esi
42 popl %ebx
43 popl %ebp
44 ret

```

图 1.31 有条件跳转指令

在代码生成时，编译器会把生成的各条代码按执行顺序划分在不同的代码块中，每个代码块相当于一个原子，其中的代码要么都不执行，要么都执行，因此这样的代码块被称为基本块。每个基本块会有一个标号作为其名字，例如图 1.31 第 25 行的 “.BB2”，这些标号通常会成为有条件或无条件跳转指令的跳转目标。

图 1.31 中第 21 行先用 cmp 指令比较变量 a 是否等于 0，若不相等，则跳转到基本块 “.BB2” 处；如果相等，则执行第 24 行的指令，而第 23 行只是个标号。图 1.31 第 34 行的指令 jle 是 Jump if Less or Equal 的缩写。第 32 至 36 行的功能是：如果 a 小于等于 b，则跳转到基本块 “.BB6”；否则(即 a 大于 b)，则执行第 36 行。常用的条件跳转指令如下表所示：

表 1.1 常用的条件跳转指令

助记符	含义	C 运算符	适用范围
JE	Jump if Equal	==	有符号或无符号数
JNE	Jump if Not Equal	!=	有符号或无符号数
JG	Jump if Greater	>	有符号数
JA	Jump if Above	>	无符号数
JL	Jump if Less	<	有符号数
JB	Jump if Below	<	无符号数
JGE	Jump if Greater or Equal	>=	有符号数
JAE	Jump if Above or Equal	>=	无符号数
JLE	Jump if Less or Equal	<=	有符号数
JBE	Jump if Below or Equal	<=	无符号数

我们以 0xFFFFFFFF 和 0x00000000 为例来进行说明，如果这两个数都被当作有符号数，

则补码 0xFFFFFFFF 是 -1，而 0x00000000 为 0，此时 -1 要小于 0；但如果都视为无符号数，则 0xFFFFFFFF 要大于 0。用于两个整数比较的指令一般都用 cmp，指令 cmp 实际进行的运算是整数的减法。但是，如果进行有符号整数之间的比较，C 编译器在产生 cmp 指令后，会选择有符号数的条件跳转指令，如表 1.1 中的 JG、JL、JGE 和 JLE；而如果要做无符号整数之间的比较，C 编译器在产生 cmp 指令后，会选择 JA、JB、JAE 和 JBE 等指令。图 1.32 给出了一个简单的测试程序，运行该程序，可以发现第 9、12 和第 15 行的 printf 语句会被执行，而第 6 行的 printf 语句没有被执行。在图 1.32 第 5 行，我们进行的是有符号整数之间的比较，此时 “-1 大于 0” 不成立。而第 8 行进行的是无符号整数之间的比较，此时无符号数 “0xFFFFFFFF 大于 0” 是成立的。我们还注意到，第 11 行和第 14 行的条件是成立的，当进行有符号整数和无符号整数之间的运算时，signed int 会被提升为 unsigned int。

```

1 #include <stdio.h>
2 int a = 0xFFFFFFFF;
3 int b = 0;
4 int main(int argc, char *argv[]){
5     if((signed int)a > (signed int)b){
6         printf("(signed int)a > (signed int)b \n");
7     }
8     if((unsigned int)a > (unsigned int)b){
9         printf("(unsigned int)a > (unsigned int)b \n");
10    }
11    if((signed int)a > (unsigned int)b){
12        printf("(signed int)a > (unsigned int)b \n");
13    }
14    if((unsigned int)a > (signed int)b){
15        printf("(unsigned int)a > (signed int)b \n");
16    }
17    return 0;
18 }
19

```

图 1.32 整数比较

我们很少会写出形如图 1.32 第 11 行这样的代码，但是如果一不小心就可能写出如图 1.33 所示的程序，其运行结果竟然是 “-1 > 0”。因此，有符号整数与无符号整数还是尽量不要混合在一起处理，其结果可能出乎意料。由图 1.33 第 22 行的 jbe 指令可以发现，此处是把 a 和 b 都当作无符号来比较了。需要注意的是，不论是把 a 当有符号数，还是无符号来比较，a 中的值始终都是 0xFFFFFFFF。

```

1 #include <stdio.h>
2 int a = -1;
3 unsigned int b = 0;
4 int main(int argc,char * argv[]){
5     if(a > b){
6         printf("-1 > 0 \n");
7     }
8     return 0;
9 }
10
11 main:
12 pushl %ebp
13 pushl %ebx
14 pushl %esi
15 pushl %edi
16 movl %esp, %ebp
17 subl $8, %esp
18 .BB0:
19 movl a, %eax
20 movl %eax, -4(%ebp)
21 cmpl b, %eax
22 jbe .BB2
23 .BB1:
24 leal .str0, %eax
25 pushl %eax
26 call printf
27 addl $4, %esp
28 .BB2:
29 movl $0, %eax
30

```

图 1.33 有符号和无符号整数

在这一小节中，我们有意地忽略了 float 和 double 类型的浮点数。原理上，CPU 只要实现了整数的加减乘除，就可以在整数运算的基础上，进行浮点数的加减乘除，这被称为浮点运算的软件实现。软件实现浮点运算的优点是可减少硬件成本，在单片机等对成本很敏感的场合，经常能看到由软件实现的浮点数运算。而在稍高端的计算机系统中，浮点运算一般都会由专门的浮点运算芯片来实现，例如 Intel 的 x87 芯片，就是一个典型的浮点运算芯片(Floating Point Unit)，缩写为 FPU。在下一小节中，我们会讨论在 UCC 编译器用到的浮点运算指令。

### 1.5.3 浮点数的算术运算

让我们还是从浮点数的算术运算入手。图 1.34 给出了一个进行浮点数运算的 C 程序。

```

1 #include <stdio.h>
2 float a = 1.0f, b = 2.0f, c = 3.0f;
3 int d;
4 int main(int argc, char * argv[]) {
5     c = a + b;
6     c = a * b;
7     c++;
8     d = *((int *)&a);
9     printf("%d %x\n", d, d);
10    d = (int) a;
11    printf("%d %x\n", d, d);
12    return 0;
13 }
```

图 1.34 浮点数运算

执行该程序，通过图 1.34 第 9 和第 11 行的输出语句，我们会得到以下结果：

```

iron@ubuntu:1.5$ ucc -o fpu fpu.c
iron@ubuntu:1.5$ ./fpu
1065353216 3f800000
1 1
```

由此结果可以发现，浮点数 1.0f 存放在内存时对应的十六进制值为 0x3f800000，这个值是按照“IEEE 754”单精度浮点数格式进行编码的，对应的十进制值为 1065353216。图 1.34 第 5 行对应的汇编代码在图 1.36 的第 12 至第 14 行。在图 1.36 中，我们删去了与 printf 相关的汇编代码，先把关注的焦点放在浮点数运算上。图 1.36 第 12 行的 fld 指令是 Load Floating point value 的助记符，用于把浮点数 a 从内存加载到 x87 浮点芯片中，指令 flds 的后缀 s 用于指示要加载的是一个单精度的浮点数。由芯片数据手册《Intel 64 and IA-32 Architectures Software Developer's Manual》，我们可知 x87 内部提供了 8 个寄存器，每个寄存器可存放 80 位的数据，足够处理 C 语言中的 32 位的 float 和 64 位的 double 类型的浮点数，这 8 个寄存器构成了如图 1.35 所描述的寄存器栈。图 1.35 中的 top 指针指向当前栈顶，在汇编代码中可用 st(0) 来表示栈顶的浮点寄存器。当使用 fld 指令从内存加载浮点数时，需要进行入栈操作。

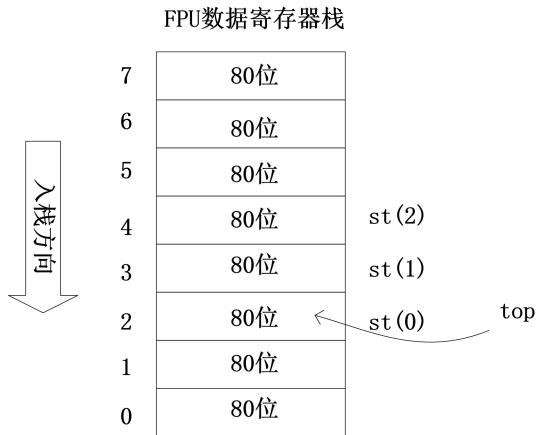


图 1.35 x87 浮点芯片的数据寄存器栈

图 1.36 第 13 行的 fadds 指令对 x87 栈顶寄存器和内存中浮点数 b 进行相加，并把结果仍旧保存在当前 x87 栈顶中。指令 fadds 的后缀 s 表示进行的是单精度的浮点数加法；若要进行 double 类型的浮点运算，则使用后缀 l，如 faddl 指令。第 14 行的 fstp 指令则用于把栈顶元素出栈，并存放到变量 c 所对应的内存单元，fst 是 STore Floating point value 的助记符，后缀 p 是 pop 的缩写。而图 1.36 第 16 行的 fmuls 则用于单精度的浮点数乘法，第 18 行的 fldl 的后缀为阿拉伯数字“1”，而非英文字母“1”，用于把浮点数 1.0 入栈。而图 1.34 第 8 行的“d = \*((int \*) &a);”实际上告诉 C 编译器，C 程序员一定要把浮点数 a 所占的 4 个字节当作一个整数来对待，此处并不作任何的编码转换，对应的汇编代码如图 1.36 第 22 至 23 行所示。我们还注意到，图 1.36 第 5 行用于初始化浮点数 a 的值，正是我们在前面看到的 1065353216。

```

1 # Code auto-generated by UCC           19 fadds c
2                                         20 fstps c
3 .data                                21
4 .globl a                               22 movl a, %eax
5 a: .long    1065353216                23 movl %eax, d
6 .globl b                               24
7 b: .long    1073741824                25 flds a
8 .globl c                               26 subl $16, %esp
9 c: .long    1077936128                27 fnstcw (%esp)
10                                         28 movzwl (%esp), %eax
11 .text                                29 orl $0x0c00, %eax
12 flds a                               30 movl %eax, 4(%esp)
13 fadds b                             31 fldcw 4(%esp)
14 fstps c                            32 fistpl 8(%esp)
15 flds a                               33 fldcw (%esp)
16 fmuls b                            34 movl 8(%esp), %eax
17 fstps c                            35 addl $16, %esp
18 fldl 1                               36 movl %eax, d

```

图 1.36 浮点数运算的汇编代码

图 1.34 的第 10 行“d = (int) a;”实际上告诉 C 编译器，C 程序员想先把浮点数 a 转换成整数。由于浮点数和整数在编码上是不一样的，我们可通过 x87 浮点处理器来实现编码转换，与之对应的汇编代码在图 1.36 第 25 至 36 行。

要理解这几行汇编代码，我们需要介绍一下 x87 控制寄存器（x87 FPU Control Word），这是一个 16 位的控制寄存器，通过改变这个寄存器的内容，可以控制 x87 芯片的行为。自然界中的实数是有无穷多个的，但能在计算机中表示的浮点数只是其中的一部分，在进行浮

点数运算时，需要做一些类似于“四舍五入”的“舍入（rounding）”操作。Intel 为 x87 芯片提供了的 4 种舍入操作，UCC 编译采用的是“截断方式（Round toward zero）”，即把超出精度的部分直接截断。我们需要在浮点运算前对 x87 控制寄存器进行设置，从而选择相应的“舍入方式”。我们还期望在完成浮点运算后，x87 控制寄存器能恢复到运算前的状态，这需要在进行浮点运算前，先在内存中保存 x87 控制寄存器的内容。通过 x87 芯片把浮点数转换成整数，该整数也需要作为一个临时变量存放到栈中。图 1.36 第 26 至 36 行的汇编代码完成了这些功能。

图 1.36 第 25 行用于把浮点数  $a$  从内存加载到 x87 芯片中，第 26 行通过调整  $esp$  寄存器，在内存栈中预留出 16 字节的空间，用于存放 x87 控制寄存器的内容和转换后的整数结果。

图 1.36 第 27 行通过 `fNSTCW` 指令把 x87 控制寄存器的内容存放到位于内存栈顶(% $esp$ )处，`fNSTCW` 是 STore x87 Control Word 的助记符，指令 `fNSTCW` 中的  $n$  表示不对浮点运算异常进行检查。图 1.36 第 28 行的 `MOVZWL` 是 MOVe with Zero-extended 的缩写，后缀  $wl$  表示把 16 位的整数扩展为 32 位的整数，相当于在高 16 位补 0。图 1.36 第 27 至 31 行用于设置 x87 控制寄存器的“舍入方式选择位（Rounding Control）”，从而选择“截断方式（Round toward zero）”的舍入操作。第 31 行的 `FLDCW` 指令用于从内存中加载控制字到 x87 控制寄存器。第 32 行的 `FISTPL` 指令实现了把 x87 栈顶寄存器中的浮点数转换成 32 位整数的操作，`FIST` 是 STore Integer 的助记符，后缀  $p$  表示在完成转换后要把 x87 栈顶寄存器出栈，而后缀  $l$  表示要把浮点数转换成 32 位整数。图 1.36 第 33 行用于恢复浮点运算前的 x87 控制寄存器，第 34 至 36 行从 8(% $esp$ )处取转换后的 32 位整数结果，并赋值给变量  $d$ ，并恢复  $esp$  指针。把浮点数转换成整数功能的汇编指令是第 32 行的 `FISTPL`，其余的汇编指令用于操作 x87 控制寄存器。

图 1.37 描述了上述过程及 x87 控制寄存器，其中 CR1 表示在浮点运算前 x87 控制寄存器中的内容，而 CR2 表示 CR1 与 0x0C00 进行“按位或”的结果，通过 CR2 我们可以设置 x87 控制寄存器的 RC 位为全 1，从而使 x87 芯片采用“截断方式（Round toward zero）”进行舍入操作。

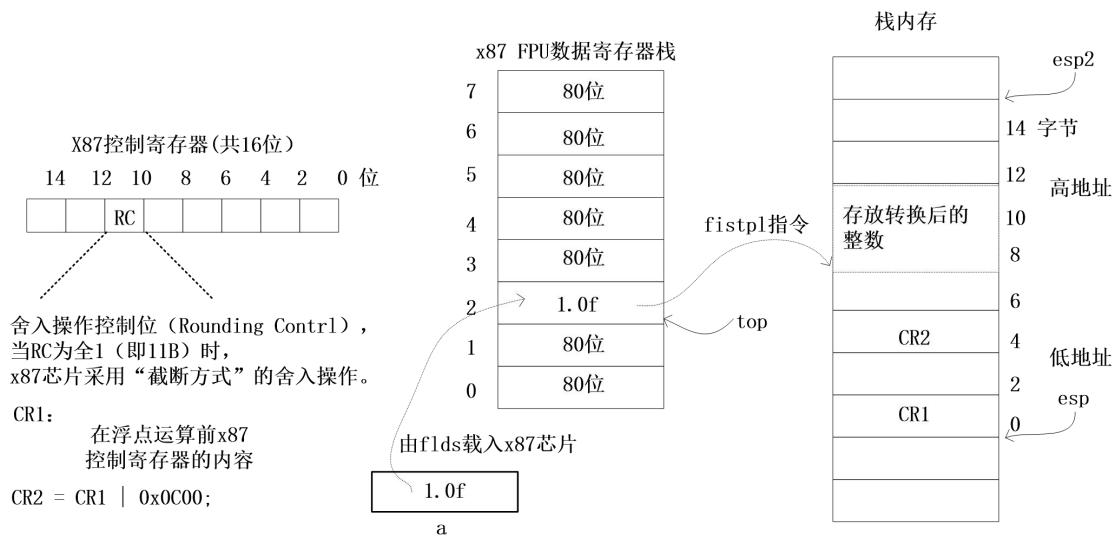


图 1.37 由 x87 实现浮点数到整数的转换

## 1.5.4 浮点数之间的比较操作

在这一小节中，我们要讨论浮点数之间比较大小的操作，如图 1.38 第 14 至 20 行所示。IEEE754 引入了一些特殊的浮点数编码来表示正无穷大、负无穷大，由一个正浮点数除以 0 可得到正无穷大，而由一个负浮点数除以 0 则可得到负无穷大。在实数范畴内，我们是不能对一个负数进行开根号的，其运算结果已经“不是一个实数”，因此 IEEE754 也引入了一个

称为“Not A Number”的特殊编码，简写为 NaN。

```

1 #include <stdio.h>
2 #include <math.h>
3 float a = 10.0f, b = 0.0f;
4 int main(int argc, char * argv[]) {
5     a = -10.0f;
6     a = a/b;
7     printf("%f\n", a);
8     a = 10.0f;
9     a = a/b;
10    printf("%f\n", a);
11    a = -16.0f;
12    a = sqrt(a);
13    printf("%f\n", a);
14    if(a > b) {
15        printf("a > b\n");
16    } else if(a == b) {
17        printf("a == b\n");
18    } else if(a < b) {
19        printf("a < b\n");
20    }
21    return 0;
22 }
```

图 1.38 NaN 和 Inf

运行图 1.38 的代码，我们会得到以下结果，图 1.38 第 7 行打印出来的结果是个负无穷大“-inf”，而第 10 打印出来的结果是正无穷大“inf”，但第 13 行打印出来的结果是 NaN。

```

iron@ubuntu:1.5$ gcc nan.c -o nan -lm
iron@ubuntu:1.5$ ./nan
-inf
inf
-nan
iron@ubuntu:1.5$ ucc nan.c -o nan
iron@ubuntu:1.5$ ./nan
-inf
inf
-nan
```

我们还注意到，图 1.38 第 14 至 20 行之间的三条 printf 语句都没有被执行。这与我们从两个整数之间比较中得到的直觉相违背。两个整数之间的比较，其最基本的结果要么是大于，要么是小于，要么是等于，三者中一定有一个成立。但通过图 1.38 的例子，我们竟然发现，两个浮点数之间的比较竟然存在“大于、小于和等于这三者都不成立的情况”。例如， $\sqrt{-16.0}$  的结果应是复数  $4i$ ，其结果已经是复数范畴，并不属于实数范畴，而 IEEE 754 浮点数标准是对实数进行编码的。一个复数对应平面上的一个点，平面上的两个点之间（两个复数之间）是不存在大小关系的。当然，我们可以比较两个复数与坐标原点之间的距离，但这需要对运算符  $>$ 、 $<$  或  $==$  进行重载。在实数这个集合中进行开根号运算，其结果可能会超出实数范围，这也是 IEEE754 引入 NaN 的出发点。换言之，在实数上的开根号运算不是封闭的，其结果可能不是实数（Not A Number）。因此两个浮点数比较的结果会“大于”、“小于”、“等于”和“无序（Unordered）”这 4 种基本情况，当其中一个浮点数为 NaN 时，就不存在大小关系。

一般而言，芯片内部的寄存器可分为控制寄存器、数据寄存器和状态寄存器这三大类。

- (1) 程序员通过设置控制寄存器的值，来发送命令或者改变芯片的行为。
- (2) 数据寄存器中一般用于存放操作数，如前文所述的 x87 数据寄存器栈，有时也会把芯片的片内地址送到数据寄存器中，因此有些芯片手册中也会引入地址寄存器的概念。
- (3) 状态寄存器则反映了芯片当前所处的状态，例如芯片是否处于 BUSY 状态。x86 CPU 内部就有一个名为 EFLAGS 的状态寄存器，我们在使用 cmp 指令进行整数之间大小比较时，会改变 EFLAGS 寄存器中的标志位。有条件跳转指令也是根据 EFLAGS 中的标志位去跳转。在前文中，我们有意忽略了 EFLAGS 寄存器的标志位，因为完全可以站在汇编指令的语义上来理解有符号和无符号整数之间的比较，就如表 1.1 所述，如果需要对 EFLAGS 寄存器做更深入理解，可查看 Intel 的 CPU 手册。

浮点单元 x87 内部也有相应的状态寄存器，而进行浮点数比较后的结果主要保存 x87

状态寄存器的 C0、C2 和 C3 这三位中，如图 1.39 所示。

x87状态寄存器

	14	13	12	10	9	8	4	0
	C3			C2	C1	C0		

x87栈顶寄存器st(0)与st(1)比较的结果

	C3	C2	C0	“C2 和 C0” 非 0 的位数及奇偶性	
st(0)>st(1)	0	0	0	0	偶
st(0)<st(1)	0	0	1	1	奇
st(0)=st(1)	1	0	0	0	偶
unordered	1	1	1	2	偶

图 1.39 x87 状态寄存器与浮点数比较的结果

有了这些概念后，我们再举一个浮点数比较的例子及其对应的汇编代码，如图 1.40 所示。图 1.40 第 12 至 13 行用于把浮点数 a 和 b 从内存中加载到 x87 数据寄存器栈的栈顶。第 14 行对处于 x87 数据寄存器栈顶的两个浮点数进行比较，fucompp 是 COMpare Floating point values 的助记符，后缀 pp 表示在浮点数比较结束后，要把栈顶的两个浮点数出栈。而指令 fucompp 中的 u 是 unordered 的缩写，表示该指令允许两浮点数比较时出现“无序（unordered）”的结果。通过图 1.40 第 15 行的指令 fnstsw，我们把 x87 状态寄存器的值传送到了 x86 的 16 位寄存器 ax 中，寄存器 ax 实际上是 32 位寄存器 eax 的低 16 位，而 8 位寄存器 ah 是寄存器 ax 的高 8 位。指令 fnstsw 是 STore x87 Status Word 的助记符，其中 n 表示“不对浮点运算异常进行检查”。第 16 行的 test 指令进行的是“按位与”运算，\$0x5 对应的是标志位 C2 和 C0。

```

1 float a = 10.0f, b = 20.0f;
2 int c;
3 int main(int argc, char * argv[]) {
4     if(a > b){
5         c++;
6     }
7     a = (float)c;
8     return 0;
9 }
10
11 .BB0:
12 flds a
13 flds b
14 fucompp
15 fnstsw %ax
16 test $0x5, %ah
17 jp .BB2
18 .BB1:
19 incl c
20 .BB2:
21 pushl c
22 fildl (%esp)
23 fstps a
24 addl $4, %esp

```

图 1.40 浮点数比较

由图 1.40 第 12 和 13 行的加载顺序可知，浮点数 a 对应的是 st(1) 寄存器，而浮点数 b 对应的是 st(0) 寄存器。在图 1.39 所示的四种比较结果中，“st(0)<st(1)”（此处就是  $b < a$ ，即  $a > b$ ）与其他三种比较结果不同的地方在于：图 1.40 第 16 行的 test 指令执行之后，得到的“标志位 C2 和 C0 中非 0 的位数”为奇数个。而图 1.40 第 17 行的汇编代码“jp .BB2”，表示当“标志位 C2 和 C0 中有偶数个 1”时，跳往基本块“.BB2”；否则执行第 19 行的汇编代码，这意味着第 4 行的 if 条件成立，我们要执行第 5 行的“c++；”。

而图 1.40 第 21 至 24 行实现了把整数转换成浮点数的运算，第 22 行的指令 fildl 是 Load Integer 的助记符，后缀 l 表示要加载的是 32 位操作数。与图 1.39 中其他的比较结果对应的汇编代码可参见文件 ucl\X86Linux.tpl，基本原理与“st(0)<st(1)”类似，都是根据 C3、C2 和 C0 这三个标志位来判断，我们就不再啰嗦。

## 1.5.5 指针、数组和结构体

图 1.41 所示的第 1 至 15 行的 C 程序包含了数组、指针和结构体，第 16 至 36 行是由 UCC 编译器编译后产生的主要汇编代码。

```

1 int * ptr;
2 typedef struct{
3     int date;
4     int month;
5     int year;
6 }Data;
7 Data dt;
8 int main(int argc,char * argv[]){
9     int number[16] = {2015};
10    ptr = &number[1];
11    *ptr = 2016;
12    dt.year = *ptr;
13    return 0;
14 }
15
16 .BB0:
17    pushl $64
18    pushl $0
19    leal -64(%ebp), %eax
20    pushl %eax
21    call memset
22    addl $12, %esp
23
24    movl $2015, -64(%ebp)
25    leal -64(%ebp), %eax
26    addl $4, %eax
27    movl %eax, ptr
28
29    movl ptr, %ecx
30    movl $2016, (%ecx)
31
32    movl ptr, %ecx
33    movl (%ecx), %edx
34
35    movl %edx, dt+8
36

```

图 1.41 数组、指针和结构体

按照 C 的语义，图 1.41 第 9 行的 C 代码是对局部数组 number 的初始化，需要把 number[0] 初始化为 2015，而数组中的其他元素则被初始化为 0。UCC 编译器采取的翻译方法是：先调用 memset 函数来把数组 number 所占的内存空间清 0，然后再把 number[0] 设为 2015，如图 1.41 第 17 至 24 行所示。C 库函数 memset 的接口如下所示：

```
void *memset(void *s, int c, size_t n);
```

按照 C 调用约定，参数需要从右到左入栈，即先要把 n 入栈，再把 c 入栈，之后才是目标地址 s。图 1.41 第 17 行的\$64 对应参数 n，表示数组 number 所占内存的大小为 64 字节，每个 int 占 4 字节，而 number 共有 16 个整数。而第 18 行的\$0 对应参数 c，表示我们要把指针 s 所指的大小为 n 字节的内存全部设为 0；第 19 行用于取内存单元-64(%ebp)的地址，第 20 行把这个地址入栈，而-64(%ebp)所对应的内存正是 UCC 编译器为局部数组 number 在栈中分配的存储空间。第 21 行完成了对库函数 memset 的调用，第 22 行把刚才入栈的 3 个参数出栈，此处每个参数正好都占 4 个字节，总共占了 12 字节的栈空间。按照 C 调用约定，这个退栈操作需要由主调函数来完成。实际上，第 17 至 22 行也演示了如何在汇编语言中调用 C 库函数。

与第 10 行 “ptr = &number[1];” 对应的汇编代码为第 25 至 27 行，第 25 行把 number[0] 的地址存到寄存器 eax 中，由于 number[1] 与 number[0] 之间的间隔为 4 字节，因此第 26 行对 eax 进行了加 4 的操作，这样 eax 中的内容就是 number[1] 的地址，第 27 行把这个地址存到了全局变量 ptr 所对应的内存单元中。

与第 11 行 “\*ptr = 2016;” 对应的汇编代码在图 1.41 的第 29 至 30 行，第 29 行通过 movl 指令把内存单元 ptr 中存放的内容传送到寄存器 ecx 中，这样 ecx 中存放的数据就是 number[1] 的地址，第 30 行的汇编指令 “movl \$2016, (%ecx)” 告诉汇编器，把 ecx 当作一个指针，把立即数 2016 存到 ecx 所指向的内存单元中，此指令执行完后，ecx 寄存器没有发生变化，但 ecx 所指向的 number[1] 内存单元被设置为 2016。第 32 至 35 行演示了“如何在汇编代码中，给结构体对象的成员域赋值”，在 C 语言中我们用的是“dt.year”，但在汇编中与之对应的是

第 35 行的“dt+8”，结构体成员域的名字 year 已经被偏移值 8 所取代。由第 2 至 6 行可知，在成员 year 的前面，还有 date 和 month 这两个成员，共占去了 8 字节，如果从 0 开始计数，则 year 正好处于第 8 个字节处。

让我们再看一份 C 程序，如图 1.42 所示。图 1.42 第 19 至 22 行的汇编代码完成了第 11 行“dt1 = dt2;”的功能，通过对汇编代码，我们可以看到寄存器 esi 和 edi 在指令“rep movsb”中的作用。图 1.42 第 19 到 22 行的汇编代码其实相当于如下 C 函数 memcpy() 的功能，寄存器 esi 相当于参数 src，寄存器名 si 是 Source Index 的缩写，起“源地址”的作用；而 edi 相当于参数 dest，寄存器名 di 是 Destination Index 的缩写，起“目的地址”的作用；寄存器 ecx 则相当于参数 n，表示要复制的字节数；真正执行复制操作的是第 22 行的汇编代码。图 1.42 第 9 行的结构体对象 dt1 由 30 个整数构成，每个整数 4 字节，共占了 120 字节，因此第 21 行指令中的立即数是 120。

```
//void *memcpy(void *dest, const void *src, size_t n);
1  typedef struct{
2    int arr1[10];
3    int arr2[20];
4  }Data;
5  typedef void FUNC(void);
6  void f(void){
7  }
8  FUNC * func = &f;
9  Data dt1,dt2;
10 int main(int argc,char **argv){
11   dt1 = dt2;
12   func();
13   (*func)();
```

```
14   f();
15   return 0;
16 }
17 .BB1:
18   leal dt1, %edi
19   leal dt2, %esi
20   movl $120, %ecx
21   rep movsb
22   call *func
23   call *func
24   call f
25 }
```

图 1.42 函数调用

图 1.42 第 12 行的“func();”和第 13 行的“(\*func)();”对应的汇编代码分别在第 23 和 24 行，都是“call \*func”。汇编指令 call 用于调用函数，“call \* func”意味着函数地址保存在变量 func 中，全局变量 func 的地址在编译连接后就可确定，但变量 func 的内容是可能发生变化的，我们需要在运行时从变量 func 对应的内存单元中取出保存在其中的函数地址。而图 1.42 第 25 行汇编指令“call f”中的函数地址 f 在编译连接后就已确定，在汇编指令中，f 实际上相当于一个地址常量。

到此，我们结合一些简单的 C 程序，把 ucc\uecl\x86Linux.tpl 中用到的有代表性的汇编代码作了介绍。

## 1.6 C 语言的变量名、数组名和函数名

对一个全局变量或静态变量 number 来讲，我们可以这样来理解出现在 C 程序中的符号 number。在 C 代码中，我们可把符号 number 理解为“number 相应内存单元中的内容”，当 number 位于赋值号右侧，表示对该内存单元进行读操作；而当 number 位于赋值号左侧时，表示对该内存单元进行写操作。C 程序员如果要获取该内存单元的地址，则使用表达式 &number。

```
// C 代码，number 对应全局静态区中的一个内存单元
number = 30;      //number 位于赋值号左侧，表示要改写 number 的内容
a = number;       //number 位于赋值号右侧，表示要读取 number 的内容
```

但在汇编代码层次，我们要把全局变量或静态变量对应的符号 number 看成是地址常数，在请求分页的操作系统中，连接器最终会为全局变量和静态变量分配一个虚地址空间中的内

存单元，相当于把汇编代码中的符号 number 替换为一个地址常数。如果要访问相应内存单元的内容，则使用如下 movl 指令；而如果要获取该内存单元的地址，可使用 leal 指令，如下所示：

```
// 若全局变量 number 的地址为 0x804a060
movl number, %eax;      // 寄存器 eax 中的内容为 30
leal number, %ebx;      // 寄存器 ebx 中的内容为 0x804a060
```

如果 number 只是个局部变量，在 C 语言层次，我们仍旧可像对待全局变量一样来理解其符号名 number。但在汇编代码层次，由于局部变量的存储空间一般会位于栈中，是动态分配的，其符号名 number 根本就不会出现在汇编代码中，而是用形如“-4(%ebp)”这样的符号来表示，其中寄存器 ebp 在运行时会指向栈空间，在编译时，我们只能算出局部变量 number 在栈中的偏移，其基地址是未知的，运行时会由寄存器 ebp 来指向。

简单来说，在 C 语言层次，对于全局变量、静态变量和局部变量，我们可以用这样的套路来理解类型为 T 的变量名 var：变量 var 占一块大小为 sizeof(T) 的内存，在 C 程序中“直接使用变量名 var”代表了 C 程序员想要读取或改变这块内存中的内容。“在变量名 var 前加一个&，即&var”表示 C 程序员想要读取这块内存对应的地址，而表达式&var 的类型为 T\*。

```
T var;
```

但这个套路并适用于函数名和数组名。按上面的套路，通过函数名 g，我们想访问的就是函数 g 所占内存中的内容，这个“内容”当然就是代码区中的代码了。然而，CPU 要执行这个函数，实际上只要知道这块代码区的首地址就可以了。基于这样的出发点，让函数名代表函数所占内存的首地址，就合情合理了。而“&g”仍可沿用上面的套路来理解。

```
void g(void);
```

对函数名 g 而言，sizeof(g) 在 C89 标准中是非法的。在编译时，sizeof 表达式是要作为一个常量来处理的，但很多时候，我们在编译一个 C 文件时，可能只在头文件中看到了函数 g 的声明，其函数定义根本就还没看到。在只知道函数接口而不知道其定义的情况下，是无法知道函数要在代码区中占多大内存的。

在 C 语言中，数组名也是个特例。如下所示，C 编译器会根据不同的上下文来对数组名进行不同的处理，这会造成语义上的不一致。这也是数组名给不少 C 程序员带来诸多困惑的源头。对以下数组 arr 来说，在 C 编译器内部的符号表中，符号 arr 的类型始终都是数组类型 int[4]，但当符号 arr 被用在不同场合时，其对应表达式的类型并不一致。

```
int arr[4];
```

(1) 数组名 arr 被用于表达式 sizeof(arr)。若 int 占 4 字节，则表达式 sizeof(arr) 的值为 16，其中的子表达式 arr 的类型是 int[4]。

(2) 数组名 arr 被用于表达式 arr+1。这里的 arr 被当成数组第 0 个元素 arr[0] 的地址，而 arr[0] 的类型为 int，表达式&arr[0] 的类型为 “int \*”，因此表达式 arr+1 中的子表达式 arr 的类型也为 int \*。

(3) 数组名被用于表达式&arr+1。子表达式 arr 的类型为数组类型 int[4]，而&arr 是指向数组 int[4] 的指针类型，即 int(\*)[4]。

我们可以大胆地猜测，C 的设计者是出于运行时效率的考虑，才会在一些情况下“把数组名 arr 当作数组元素 arr[0] 的首地址”。例如，在以下函数调用“f(bigArr);”中，若符号 bigArr 代表的是数组的内容，则在传参时我们需要传递 4000 字节的数据，这要占用较多的栈空间，同时大量数据的复制也要耗费不少时间。此时，若由 C 编译器把 f(bigArr) 中的 bigArr 当作 bigArr[0] 的地址，则只要传递一个地址就可以了，同时函数 f 的形参 int num[1000] 也可由 C 编译器隐式地调整为 int \* num。但是这并不能完全阻止 C 程序员传递数组的内容，C 程序员还是可以写出如下 struct Container，通过给函数 k 传递一个 struct Container 对象，C 编译器还是会复制其中的数组 data。

```

int bigArr[1000]; //假设 sizeof(int) 为 4
void f(int num[1000]){
}
void g(void){
    f(bigArr);
}
struct Container{
    int data[1000];
};
void k(struct Container d){
}

```

如果从语义一致上的角度出发，在 C 语言层次，让数组名 bigArr 代表数组中的内容其实也是很好的设计。这或许还更符合“提供机制，而非策略”的思想，即由 C 编译器提供传参的各种机制，而 C 程序员根据需要去选用一种，就如以下函数声明 h1 和 h2 所示。

```

void h1(int arr[1000]);
void h2(int * ptr);

```

而且如果让数组名代表数组中的内容，在进行数组对象之间的复制时，C 程序员还能写出更加简洁的表达式，如下所示：

```

int b1[1000];
int b2[1000];
b1 = b2;           //如果数组名代表数组中的内容

```

不过，当一个决定已成了标准，我们就要严格遵守。在后续分析 UCC 编译器源代码时，我们会看到编译器对函数名和数组名的特殊处理。

## 1.7 C 语言的变参函数

UCC 编译器中有不少地方使用了 C 语言的变参函数，在这一节中，我们来讨论一下 C 语言变参函数的实现原理。C 标准库中的 printf 函数就是一个典型的变参函数，其接口如下所示，函数声明中的省略号表明这是一个变参函数。

```
int printf(const char *format, ...);
```

我们先举一个简单的例子来说明 printf 函数的调用过程，如图 1.43 所示。图中第 1 至 11 行对应的是 hello.c，而第 12 至 25 行是由 UCC 编译器生成的抽象语法树 hello.ast，第 26 至 33 行则是 UCC 产生的中间代码 hello.uil，第 34 至 52 行则是 UCC 生成的部分汇编代码 hello.s。

```

1 #include <stdio.h>
2 float a = 1.0f;
3 double b = 2.0;
4 int c = 3;
5 char d = 'q';
6 int main(int argc, char * argv[]) {
7     printf(
8         "a = %f, b = %f, c = %d, d = %c\n",
9         a, b, c, d);
10    return 0;
11 }
12 // hello.ast
13 function main
14 {
15     (call printf
16         str0,
17         (cast double
18             a),
19         b,
20         c,
21         (cast int
22             d))
23     (ret 0)
24 }
25
26 // hello.uil
27 function main
28 t0 :&str0;
29 t1 : (double)(float)a;
30 t2 : (int)(char)d;

```

```

31 printf(t0, t1, b, c, t2);
32 return 0;
33 ret
34 // hello.s
35 movl %esp, %ebp
36 subl $16, %esp
37 .BB0:
38 leal .str0, %eax
39 flds a
40 fstpl -12(%ebp)
41 movsb1 d, %ecx
42 pushl %ecx
43 pushl c
44 subl $8, %esp
45 fldl b
46 fstpl (%esp)
47 subl $8, %esp
48 fldl -12(%ebp)
49 fstpl (%esp)
50 pushl %eax
51 call printf
52 addl $28, %esp

```

图 1.43 printf() 函数

我们注意到第 9 行的实参 *a* 是 float 类型，而 *d* 是 char 类型，第 17 行的抽象语法树用于把实参 *a* 从 float 转换成 double 类型，而第 21 行则用于把实参 *d* 从 char 转换成 int 类型。对于变参函数中的“无名参数”，例如上述第 9 行从 *a* 开始的参数，这些参数在函数 printf 的声明中并没有与之对应的形参名。按 C 标准的要求，C 编译器会对这些无名参数进行实参提升的操作，即把小于 int 型的 char 和 short 提升为 int 类型，把 float 类型提升为 double 类型。图 1.43 第 29 至 31 行的中间代码很直观地反映了这个实参提升的过程。与之对应的汇编代码如第 39 至 52 行所示，第 39 至 40 行的代码把 float 类型的 *a* 转换为 double 类型，转换后的结果存到临时变量-12(%ebp)中，我们在 1.5 节时介绍过与浮点运算相关的汇编指令。按照 C 调用约定，参数按从右到左的次序依次入栈，第 41 行的汇编指令完成了把参数 *d* 由 char 到 int 的转换，第 42 行把转换后的结果入栈，第 43 行则把参数 *c* 入栈，第 44 至 46 行则从全局静态数据区加载双精度浮点数 *b* 并入栈，第 47 至 49 行则从临时变量-12(%ebp)中加载双精度浮点数，并入栈。第 50 行把格式化字符串的首地址入栈，我们在第 38 行时已将其地址存到寄存器 *eax* 中，第 51 行则进行真正的函数调用。由于所有的参数都是存放在栈中，共占去了 4+8+8+4+4（即 28）字节，当被调函数 printf 返回时，主调函数 main()会在第 52 行进行参数出栈的操作，即把 esp 指针进行加 28 的操作。

库函数 printf() 的代码在我们编写上述 hello.c 时就已经存在，这意味着被调函数 printf 其实并不知道我们在 main() 中调用它时，到底传递了几个实参。对 printf 而言，它只是按照格式化字符串的说明，从栈中取出相应的参数，如下所示：

```

// 实际上只有 10 这一个参数，但 printf 看到有两个%d,
// 于是仍试图从栈中取两个参数，打印出形如 10,1074172310 的垃圾值
printf("%d, %d ", 10);
// 实际上有 10,20,30 这 3 个参数，但 printf 只看到一个%d,
// 于是只打印出参数 10
printf("%d", 10, 20, 30);

```

图 1.44 更清楚地描述了图 1.43 的内存布局，虚线左侧是栈区，右侧为全局静态数据区。在栈区中，我们标出了真正入栈的参数类型依次为 int、int、double、double 和 char \*，它们共占据了 28 个字节的栈空间。而格式化字符串实际上是存放在全局静态数据区的，压入栈中的只是该字符串的首地址。

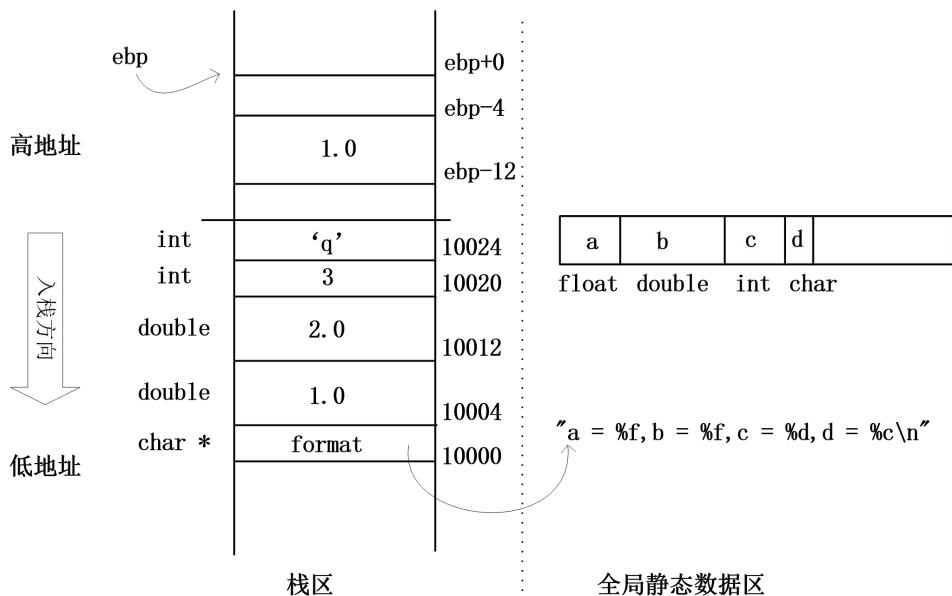


图 1.44 内存布局示意图

下面，让我们换个角度来看问题，假设我们是 `printf` 库函数的实现者，在 `printf` 函数的函数体内，我们通过形参 `format` 就可以访问到全局静态数据区中的格式化字符串，通过表达式`&format`，我们就可以知道 `format` 在栈中的内存地址。由图 1.44 所示的内存布局，通过`&format` 可计算出其他参数的地址，有了这些参数的内存地址，我们就可以访问它们。当然，不同的 C 编译器所产生的栈内存布局是有所不同的。为计算方便，让我们不妨假设`&format` 的值为十进制的 10000，由图 1.44 可算出 `format` 上方的 4 个无名参数对应的地址依次为十进制的 10004、10012、10020 和 10024。

按照这样的思路，我们可以写出如下所示的变参函数 `OurPrintfV1()`，用于从栈中取出无名参数，这个函数纯粹是为了演示如何根据形参 `format` 的地址来访问其他无名参数，如图 1.45 所示。需要注意的是，我们有意忽略了对格式化字符串的处理过程，虽然这只是一个简单的字符串判断，并不会太复杂。在函数体中添加的 `printf` 调用，只是为了验证我们确实从栈中取出了参数。

```

1 // OurPrintfV1
2 int printf(const char *format, ...);
3 float a = 1.0f;
4 double b = 2.0;
5 int c = 3;
6 char d = 'q';
7 void OurPrintfV1(const char *format, ...){
8     unsigned int addr = (unsigned int) &format;
9     addr += sizeof(char *);
10    printf("%f ",*((double *)addr));
11    addr += sizeof(double);
12    printf("%f ",*((double *)addr));
13    addr += sizeof(double);
14    printf("%d ",*((int *)addr));
15    addr += sizeof(int);
16    printf("%c \n", (char) (*((int *)addr)) );
17 }
18 int main(int argc, char * argv[]){
19     OurPrintfV1("a = %f,b = %f,c = %d,d = %c\n",a,b,c,d);
20     return 0;

```

```
21 }
```

图 1.45 OurPrintfV1()

为了让图 1.45 中的代码看起来更优雅些，我们会引入一些宏来处理“由 format 的地址来定位其他无名参数”的过程，图 1.46 中的 OurPrintfV2()完成的工作与 OurPrintfV1()类似，但看起来简洁多了。图 1.46 第 16 行的宏 va\_start(ap,format)使指针 ap 指向第一个无名参数，而第 17 行的 va\_arg(ap,double)把第一个无名参数当作 double 型取出，并使 ap 指向第二个无名参数。依此类推，不难理解图 1.46 第 18 至 20 行的其他 va\_arg 宏。第 21 行的 va\_end(ap)只是设置指针 ap 为 NULL，表示我们对指针 ap 的使用已经到此为止。

```

1 // OurPrintfV2
2 int printf(const char *format, ...);
3 float a = 1.0f;
4 double b = 2.0;
5 int c = 3;
6 char d = 'q';
7
8 typedef char * va_list;
9 #define ALIGN_INT(n) ((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))
10 #define va_start(list, start) (list = (va_list)&start + ALIGN_INT(start))
11 #define va_arg(list, t) (* (t *) ((list += ALIGN_INT(t)) - ALIGN_INT(t)))
12 #define va_end(list) (list = (va_list)0)
13
14 void OurPrintfV2(const char *format, ...){
15     va_list ap;
16     va_start(ap,format);
17     printf("%f ",va_arg(ap,double));
18     printf("%f ",va_arg(ap,double));
19     printf("%d ",va_arg(ap,int));
20     printf("%c \n", (char)va_arg(ap,int));
21     va_end(ap);
22 }
23 int main(int argc, char * argv[]){
24     OurPrintfV2("a = %f,b = %f,c = %d,d = %c\n",a,b,c,d);
25     return 0;
26 }
```

图 1.46 OurPrintfV2()

我们有意在图 1.46 中第 8 至 12 行列出 va\_arg 等宏的定义，这些宏来自于 UCC 编译器的头文件 ucl\linux\include\stdarg.h。实际使用这些宏时，我们只需要包含头文件 stdarg.h 就可以。需要注意的是，不同 C 编译器的栈内存布局是可能是不同的，例如在 64 位平台上的栈内存布局就与 32 位平台不一样，我们在图 1.45 中给出的函数 OurPrintfV1()就不能用于 64 位平台。而库函数提供的 va\_arg 等宏定义屏蔽了这些实现上的细节，虽然不同平台的 va\_arg 宏在实现上有所不同，但对 C 程序员而言，通过这些宏来访问无名参数的流程却是一致的。这也是我们要通过 stdarg.h 中的宏来访问无名参数的另一个原因。其流程如下：

- (1) 通过“va\_list ap;”定义一个用于指向无名参数的指针变量 ap。
- (2) 通过“va\_start(ap,变参函数最右边的有名参数);”使 ap 指向第一个无名参数。例如在 OurPrintfV2()函数中，最右边的有名参数就是 format。
- (3) 通过有名参数指向的字符串来确定第 1 个参数的类型 T，由 va\_arg(ap,T)取得第一个无名参数，并使 ap 指向第 2 个无名参数。当然这些字符串的格式需要我们事先约定好，例如，对 printf 函数而言，%d 对应的是 int 类型。在 OurPrintfV2()函数中，我们可以通过有名参数 format 获得相应的字符串，对字符串进行简单的解析就可取出%d 或者%f。

(4) 仿照第(3)步，通过宏 `va_arg()`依次取出其他的无名参数。

(5) 通过“`va_end(ap);`”来表示我们不再使用 `ap` 指针。

不同平台的栈内存布局有所不同，但它们实现变参函数的基本原理应是相似的。通过对图 1.44、图 1.45 和图 1.46 的分析，我们应可以触类旁通、举一反三。这也是所谓的“伤其十指，不如断其一指”。

C 语言简洁而有力，但是要较好地驾驭 C 语言，则需要对形如图 1.44 的内存布局有较清楚的理解。在很多时候，一些 C 程序员无法较好地使用 C 语言指针的原因在于“对相关内存布局没有较清晰的概念”。C++实际上也对程序员提出了类似的要求，即便是在有意淡化指针概念的 Java 语言中，如果对内存布局完全没有概念，也是有可能写出如下所示的 Java 程序。该程序员原本期待两次对 `bg.f()` 的调用能各打印出 5 条 Hello，即共 10 条 Hello，却诧异地发现一共只有 5 条 Hello。

```
class Bug{
    int i = 0;
    public void f(){
        for(;i < 5; i++){
            System.out.println("Hello");
        }
    }
    public static void main(String args[]){
        Bug bg = new Bug();
        bg.f();
        bg.f();
    }
}
```

在有些情况下，我们在形如图 1.46 的变参函数 `OurPrintfV2()` 中，只是想做一些准备工作，真正对无名参数进行访问的操作，我们还是想交由别的函数来处理，比如 `uclerror.c` 中的 `Do_Error` 函数，如图 1.47 所示。

```
1 // 错误处理
2 //
3 void Do_Error(Coord coord, const char *format, ...)
4 {
5     va_list ap;
6
7     ErrorCount++;
8     if (coord)
9     {
10         fprintf(stderr, " (%s,%d):", coord->filename, coord->ppline);
11     }
12     fprintf(stderr, "error:");
13     va_start(ap, format);
14     vfprintf(stderr, format, ap);
15     fprintf(stderr, "\n");
16     va_end(ap);
17 }
```

图 1.47 `Do_Error()`

在图 1.47 第 7 行，我们把表示错误个数的全局变量 `ErrorCount` 加 1，第 10 行则打印出错的 UCC 编译器源代码文件名和出错的行号，但处理 `Do_Error` 函数无名参数的工作，我们还是想交给库函数 `vfprintf()`，如图 1.47 第 14 行所示。对 `Do_Error` 的函数调用如下所示：

```
Do_Error(coord, "struct member %s doesn't exist", "abc");
```

结合图 1.44 可知，通过预先约定的格式化字符串，我们可以知道入栈的无名参数的个数及其类型，结合第 1 个无名参数的首地址，我们就可以访问在栈中的无名参数了。图 1.47 第 14 行的函数调用“vfprintf(stderr, format, ap);”的实参中就有指向格式化字符串的 format 和指向第 1 个无名参数的 ap。库函数 vfprintf() 的接口如下所示：

```
int vfprintf(FILE *stream, const char*format2, va_list ap2);
```

图 1.48 给出了调用函数 Do\_Error() 时的内存示意图，从图中我们可以看到，vfprintf 的形参 ap2 已经指向了 Do\_Error 函数的无名参数的开始位置，函数 vfprintf 的形参 format2 则指向了格式化字符串。对 vfprintf 函数而言，有了格式化字符串的首地址，且又有了无名参数的首地址，七颗龙珠已经凑齐，可以召唤神龙了。

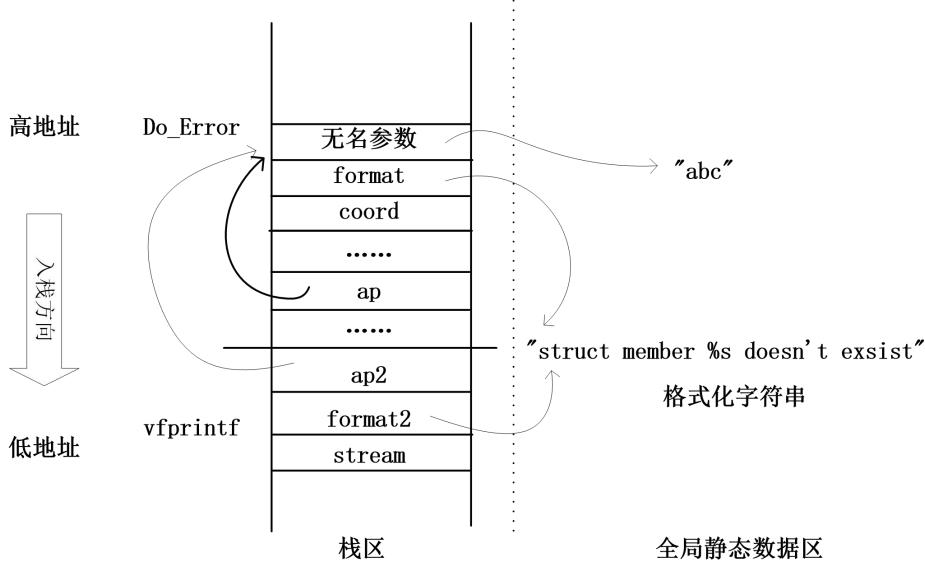


图 1.48 Do\_Error 的栈示意图

回顾整个第 1 章，我们从语言、递归和文法入手，通过一个简单的例子 ucc\examples\sc，介绍了如何由文法来编写语法分析器，如何在生成的语法树上进行“中间代码生成”。由于在后续章节剖析 UCC 编译器时，还需要在中间代码的基础上产生汇编代码，因此在第 1.5 节，我们还介绍了 UCC 编译器所用到的主要汇编代码。

经过第 1 章的热身运动，我们准备开始正式踏上 UCC 编译器剖析的漫漫长征路。所谓“工欲善其事，必先利其器”，在 Ubuntu 环境下可先安装 wine，然后安装 Source Insight 等 Windows 平台的代码阅读工具。这种环境可能要被 Linux 的忠实爱好者鄙视，但从简单实用的角度出发，不失为一种不错的选择。边阅读 UCC 的源代码，边加上自己的一些注释和体会，看得兴起之时，再加上一些用于调试的代码，例如使用 ucc\ucl\ucl.h 中的 PRINT\_DEBUG\_INFO 等宏定义来输出调试信息，观察一下编译器的运行结果是否如我们所预期。如果感觉自己似乎发现了潜在的问题，就写一些简单的测试代码用来触发 UCC 编译器的 Bug。慢慢的，我们就会进入正轨了。世界上不存在完全健康的人，同样道理，稍复杂点的软件，或多或少都存在 Bug。把软件产品改到没有 bug 再发布，几乎是不可能的。不然，Windows 也不会经常需要打补丁了。诚如台湾作家龙应台所言，“所谓了解，就是知道对方心灵最深的地方的痛处，痛在哪里”。相信随着阅读的深入，您会发现 UCC 编译器的一些 Bug。目录 ucc\examples 中包含了已发现的 Bug，例如在文件 ucc\examples\array\array.c 的开头，包含了以下注释，其含义是 array.c 的代码是针对 ucc 编译器 exprchk.c 中的函数 CheckPostfixExpression()。

```
/*
 * see CheckPostfixExpression(AstExpression expr) in exprchk.c
 */
```

最后，让我们以下面这段代码来结束第一章。再好的书，再好的资料，最多只能帮助我们少走弯路，让我们在无助时有个依靠。但是“有些事，只能一个人做。有些关，只能一个人过。有些路啊，只能一个人走”。

```
while(还没读懂 ucc\examples\sc) {
    阅读并上机运行 ucc\examples\sc 的代码
}
```

## 1.8 本章习题

1. 请为第 1.3 节中的 ucc\examples\sc 添加 do 语句，并仿照 while 语句，为 do 语句生成相应的中间代码。Do 语句的产生式如下所示：

DoStatement:

```
do Statement while ( Expression );
```

2. 请编写一个实现以下功能的变参函数 myprintf()。

```
*****
```

```
int myprintf(const char * fmt, ...);
    @param const char * fmt 格式化字符串
    @return 返回值代表总共输出了几次整数
```

使用示范：

```
int a = 5; int b = 3; int c;
c = myprintf("%1 + %2 = %3, %1 - %2 = %4\n", a,b,a+b,a-b);
```

运行结果：

$5 + 3 = 8, 5 - 3 = 2$

//c 的值为 6，表示共输出了 5、3、8、5、3 和 2 这 6 个整数。

使用说明：

其中的%1 代表变参函数 myprintf() 的第 1 个无名参数。

%n 代表变参函数 myprintf() 的第 n 个无名参数，为简单起见，我们规定 n 不超过 9，且所有的无名参数都是 int 型。

```
*****
```

3. 请从变参函数的实现原理出发，解释如下所示的“va\_arg(ap,float)”存在什么问题。

```
#include <stdarg.h>
void OurPrintf(const char *format, ...) {
    va_list ap;
    va_start(ap, format);
    printf("%f ", va_arg(ap, float));
    //略
}
```

4. 编写一个求 Fibonacci 数列的 C 程序，用 UCC 编译器来编译并运行这个程序，结合该 C 程序源代码，分析由 UCC 生成的语法树、中间代码和汇编代码。

5. 请仿照图 1.12，结合 ucc\examples\sc 中的源代码，画出以下声明对应的分析树和语法树。

```
int (*arr)[3][5];
```

# 第 2 章 UCC 编译器的基本模块

## 2.1 从 Makefile 起

目录 `ucc\ucl` 包含了 UCC 编译器的源代码，而 `ucc\driver` 则是 UCC 驱动的源代码，我们在第 1 章时已简要地分析过 UCC 驱动。从第 2 章开始，我们要剖析的代码就集中在目录 `ucc\ucl` 中，`ucl` 就是 UCC 编译器的名称。我们将按照词法分析、语法分析、语义检查、“中间代码生成及优化”和“汇编代码生成”这样的脉络来剖析 UCC 编译器的源代码。习惯上，我们会先看看 `Makefile` 文件。Linux 平台上对应的 `Makefile` 文件如图 2.1 所示。

```

1 C_SRC      = alloc.c ast.c decl.c declchk.c dumpast.c emit.c \
2           error.c expr.c exprchk.c flow.c fold.c gen.c \
3           input.c lex.c output.c reg.c simp.c stmt.c \
4           stmntchk.c str.c symbol.c tranexpr.c transtmt.c type.c \
5           ucl.c uildasm.c vector.c x86.c x86linux.c
6 OBJS       = $(C_SRC:.c=.o)
7 CC         = gcc
8 CFLAGS     = -g -D_UCC
9 UCC        = ./driver/ucc
10
11 all: $(OBJS) assert.o
12 $(CC) -o ucl $(CFLAGS) $(OBJS)
13
14 clean:
15 rm -f *.o ucl
16
17 test: $(C_SRC)
18 $(UCC) -o ucl1 $(C_SRC)
19 mv $(UCCDIR)/ucl $(UCCDIR)/ucl.bak
20 cp ucl1 $(UCCDIR)/ucl
21 $(UCC) -o ucl2 $(C_SRC)
22 mv $(UCCDIR)/ucl.bak $(UCCDIR)/ucl
23 strip ucl1 ucl2
24 cmp -l ucl1 ucl2
25 rm ucl1 ucl2
26

```

图 2.1 Makefile

在图 2.1 中，第 11 行的 `all`、第 14 行的 `clean` 和第 17 行的 `test` 是 make 的三个目标，其中第 11 行的 `all` 是缺省的目标，图 2.1 第 12 行用 GCC 编译器来编译 UCC 的源代码，并生成 UCC 编译器 `ucl`。第 14 行的目标 `clean` 用于删除编译时产生的目标文件和可执行程序，第 17 至 25 行的目标 `test` 通过 UCC 编译器的自举（bootstrap）来做测试，用于测试的代码就是 UCC 编译器本身的源代码。图 2.2 表述了“make all”和“make test”的主要功能。

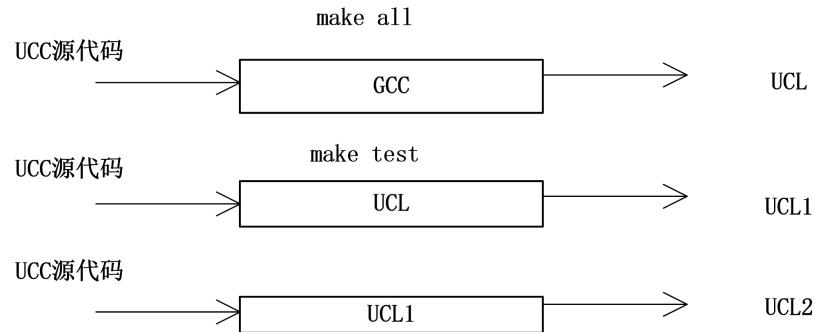


图 2.2 make all 和 make test

由图 2.2，我们看到一个很有趣的现象，UCC 源代码经 GCC 编译后产生的可执行程序 ucl 本身就是一个 C 编译器。我们可以用 ucl 再来编译 UCC 源代码，得到一个新的编译器 ucl1，接着还可以用 ucl1 来编译 UCC 源代码，生成编译器 ucl2。这相当于 UCC 实现了“自我编译”，其源代码还起到了“测试代码”的作用。当然，这种测试是不完全的，不代表 UCC 源代码就不存在 Bug 了。例如，UCC 编译器虽然可以为浮点运算产生相应的汇编指令，但在其自身 C 源代码中，几乎不存在浮点数运算。因此，在 UCC 自举的测试过程，并没有对浮点运算进行测试。限于人力和物力，UCC 编译器在测试方面所做的工作还不够。

接下来，让我们看一下 UCC 编译器的 main() 函数，如图 2.3 所示，第 6 行用于处理命令行参数，第 8 行用于初始化跟寄存器管理有关的数据结构，第 9 行则是词法分析器的初始化，而第 10 行进行类型系统的初始化，我们会在稍后再对这些用于初始化的函数进行讨论。这里，我们主要通过 main() 函数，熟悉一下 UCC 编译器的内部工作流程。

```

1 int main(int argc, char *argv[]){
2     int i;
3
4     CurrentHeap = &ProgramHeap;
5     argc--; argv++;
6     i = ParseCommandLine(argc, argv);
7
8     SetupRegisters();
9     SetupLexer();
10    SetupTypeSystem();
11    for (; i < argc; ++i){
12        Compile(argv[i]);
13    }
14
15    return (ErrorCount != 0);
16 }
  
```

图 2.3 UCL main()

图 2.3 第 11 至 13 行的 for 循环，用于依次编译在命令行中出现的各个 C 文件。严格说来，UCC 驱动传递给 UCC 编译器的文件是经预处理后的文件，而不是原始的 C 文件。这里，为简单起见，我们在表述时就把 C 文件当作 C 编译器的输入。实际上，我们在使用 UCC 编译器时，一般都是通过编译器驱动命令 ucc，很少会直接使用命令 ucl。与编译相关的主要工作是在图 2.3 第 12 行的 Compile() 函数中实现。图 2.4 给出了 Compile() 函数的代码。

```

1 static void Compile(char *file){
2     AstTranslationUnit transUnit;
3     Initialize();
4
5     transUnit = ParseTranslationUnit(file);
  
```

```

6   if (ErrorCount != 0) {
7     goto exit;
8   }
9   CheckTranslationUnit(transUnit);
10  if (ErrorCount != 0) {
11    goto exit;
12  }
13  if (DumpAST) {
14    DumpTranslationUnit(transUnit);
15  }
16  Translate(transUnit);
17  if (DumpIR) {
18    DAssemTranslationUnit(transUnit);
19  }
20  EmitTranslationUnit(transUnit);
21 exit:
22  Finalize();
23 }

```

图 2.4 Compile()

在图 2.4 的第 2 行，我们看到了一个名为 `AstTranslationUnit` 的类型，其前缀正是我们在第 1 章介绍过的抽象语法树 AST，而 `TranslationUnit` 是编译单元的意思。C 编译器把一个 C 文件视为一个独立的编译单元，当有多个 C 文件需要编译时，C 编译器是分别进行编译的。`TranslationUnit` 实际上就是 C 语言标准文法的开始符号，在《The C Programming language》书中附录“`A.13 Grammar`”里包含了完整的 C 语言文法，这本书由 Brian W.Kernighan 和 C 语言的创始人 Dennis M.Ritchie 合著，江湖人尊称为《K&R》。对照着 C 语言文法来阅读 UCC 语法分析器的源代码。我们会发现 UCC 确实是用心良苦，语法分析函数的命名几乎都是与 C 文法中的非终结符一一对应。例如在图 2.4 第 5 行调用的 `ParseTranslationUnit()`，通过函数名，我们就能知道这是用来对编译单元进行语法分析的。图 2.4 第 6 行检查一下在语法分析过程中是否出现语法错误，如果没有错误则进入下一个阶段“语义检查”，这可通过在第 9 行调用函数 `CheckTranslationUnit()` 来实现。通过语法分析，我们会构建一棵巨大的语法树，第 5 行的变量 `transUnit` 就指向这棵语法树的树根。而后续的语义检查也需要在这棵语法树上进行，因此在第 9 行调用 `CheckTranslationUnit()` 时，我们传入了参数 `transUnit`。

语义检查时，需要根据 IT 大佬们制定的标准进行，通过 <http://flash-gordon.me.uk/ansi.c.txt> 可下载 C89 标准文档 `ansi.c.txt`，在后续章节我们谈到 `ansi.c.txt` 时，都是指这份文档。C 语言历经几十年而始终屹立江湖，其标准本身也历经几次修订，比较知名的有 C89、C99 和 C11。UCC 是按 C89 进行编译器的构造。图 2.4 第 9 行就是根据 C89 标准进行语义检查。CSDN 网站上曾有一篇对 C++ 编译器早期实现者 Stanley B. Lippman 的一篇专访，Lippman 很惬意地回顾了还没有 C++ 标准委员会时，写 C++ 编译器是何等之畅快，那时基本上就是他和 C++ 创始人 Bjarne Stroustrup 商量一下，就可以上机实现。就如创业公司在只有几杆枪时，效率是何其之高，但规模变大后，就开始有政治，有江湖了。但是，产品要大规模推广，那就一定要标准化，这需要 IT 大佬们坐下来一起商量，在 `ansi.c.txt` 的前言部分我们能看到各大软件和硬件 IT 公司的大名。

如果语义检查通过，并且使用 UCC 编译器的 C 程序员在命令行中添加了参数“`--dump-ast`”，则 UCC 编译器会把经语法分析和语义检查后形成的语法树打印到文件中，图 2.4 第 14 行完成了这个功能。接下来，就进入了中间代码生成阶段，如图 2.4 第 16 行的 `Translate(transUnit)` 所示，其中的参数 `transUnit` 再次提醒我们，我们是在语法树的基础上来生成中间代码的。如果 C 程序员在命令行中指定了参数“`--dump-IR`”，则第 18 行会把

中间代码打印到文件中。第 20 行的函数 EmitTranslationUnit() 用于产生 x86 汇编代码。

以上就是 UCC 编译器的主要工作流程，每一阶段的具体过程我们会在后续章节依次进行分析。通过图 2.3 和图 2.4，我们可以在脑海中建立一个总体的轮廓。

## 2.2 词法分析

目录 `ucc\ucl` 下，与词法分析相关的 C 文件主要有 `input.c` 和 `lex.c`，`input.c` 用于从外存读入预处理后的文件，其主要的函数如图 2.5 所示。由于在 UCC 驱动的代码中，我们已经预定义了宏 `_UCC`，因此第 2 行的条件会成立，函数 `ReadSourceFile()` 会使用 C 标准库的 IO 函数来读取文件。C 源代码文件只是普通的文本文件，其文件大小很少超过 1M 字节的，现在的计算机有足够的内存来存放 C 文件，因此 UCC 编译器并没有采取逐行读的行缓存策略，而是一下把整个 C 文件读入内存中，第 4 行的 `fopen()` 用于打开文件，通过第 8 行的 `fseek()` 函数和第 9 行的 `ftell()` 函数，就可以获得文件的大小，然后通过第 11 行的 `malloc()` 开辟了足够大的内存堆空间，第 18 行通过 `fread()` 把整个文件读入内存中。

```

1 void ReadSourceFile(char *filename) {
2 #if defined(_UCC)
3     int len;
4     Input.file = fopen(filename, "r");
5     if (Input.file == NULL) {
6         Fatal("Can't open file: %s.", filename);
7     }
8     fseek(Input.file, 0, SEEK_END);
9     Input.size = ftell(Input.file);
10    // allocate enough heap memory.
11    Input.base = malloc(Input.size + 1);
12    if (Input.base == NULL) {
13        Fatal("The file %s is too big", filename);
14        fclose(Input.file);
15    }
16    // set current position to the beginning of the file.
17    fseek(Input.file, 0, SEEK_SET);
18    Input.size = fread(Input.base, 1, Input.size, Input.file);
19    fclose(Input.file);
20    //略
21 }
```

图 2.5 `ReadSourceFile()`

此后，文件 `lex.c` 中的词法分析器 `GetNextToken()` 就不再需要去读写文件了，只需要从内存中读取字符即可。头文件 `ucc\ucl\input.h` 中的结构体 `struct input` 记录了关于输入文件的信息，如图 2.6 所示，其中第 10 行的 `base` 域记录了读入的输入文件在内存中的起始位置，第 11 行的 `cursor` 则反映了当前的读取位置。例如，对一个名为 `hello.c` 的 C 文件而言，UCC 编译器见到的是经预处理后的文件 `hello.i`。文件 `hello.i` 包含了各个头文件及 C 文件 `hello.c`，为了在错误处理时能准确报告出错位置，引入了图 2.6 第 1 行的结构体 `struct coord`，`coord` 是坐标 (`coordinate`) 的缩写。图 2.6 第 2 行的 `filename` 指向出错的原始文件名，比如 `hello.c`；第 3 行的 `ppline` 记录出错的行号，这个行号指的是在 `hello.c` 中的行号；而第 4 行的 `line` 代表的是在预处理后的文件 `hello.i` 中的行号；第 5 行的 `col` 用于存放出错的列号。

```

1 typedef struct coord{
2     char *filename;
3     int ppline;
```

```

4     int line;
5     int col;
6 } *Coord;
7 // information about input file
8 struct input{
9     char *filename;
10    unsigned char *base;
11    unsigned char *cursor;
12    unsigned char *lineHead;
13    int line;
14    // file handle returned by fopen() / CreateFileA() / open()
15    void* file;
16    // handle returned by CreateFileMapping() on Win32
17    void* fileMapping;
18    // file size
19    unsigned long size;
20 };

```

图 2.6 struct input

我们在第 1 章讨论变参函数时,在图 1.47 中给出了用于进行错误处理的函数 Do\_Error()。但是在 UCC 源代码中,为了在 UCC 编译器开发调试时,能快速找到发现错误的编译器源代码所处的位置,我们并不直接使用 Do\_Error() 函数,而是使用宏函数 Error 来报错。宏 Error 的定义如下所示:

```
#define Error fprintf(stderr, "(%s,%d):",FILE,LINE),Do_Error
```

通过 Error(),我们可以得到如下所示的出错提示,其中的“(hello.c,3)”表示 C 程序员编写的 hello.c 的第 3 行出现错误,而“(stmt.c 24)”则表示我们是在 UCC 源代码 stmt.c 的第 24 行检测到这个错误的。

```
(stmt.c 24):(hello.c,3):error:Expect ;
```

当然,真正把 UCC 编译器作为一个产品发布时,“(stmt.c 24):”这样的信息是没有必要打印出来的。此时,只要把 Error 的宏定义改为“#define Error Do\_Error”即可。接下来,我们来分析一下 lex.c 中用于词法分析的代码。在 C 语言中,一个 char 字符对应一个字节,一个字节包含了 0 至 255 种不同的数值,若不考虑效率,我们可以编写一个 switch 语句,根据当前字符的值来分门别类地进行处理,如下所示。

```
switch(*cursor){
    case 0: ... ; break;
    case 1: ... ; break;
    ...
    case 255:...; break;
}
```

不过,当遇到一个 switch 语句中有这么多的 case 时,我们往往会采取“打表”的方法来简化,如图 2.7 第 10 行的 Scanners[256]数组所示。每调用一次第 12 行的函数 GetNextToken(),我们就会获得一个由若干个字符构成的单词,第 16 行的 SkipWhiteSpace() 跳过了注释、空格、制表符、回车和换行等空白符,真正进行词法分析的是第 19 行的代码。图 2.7 第 3 至 8 行的枚举常量中包含了 UCC 编译器用到的各种 token 的类别,例如 TK\_CONST 和 TK\_TYPEDEF 等,这些枚举常量都被放在第 6 行所示的文件 token.h。当词法分析器 GetNextToken() 读取一个单词后,会在第 20 行会返回其 token 类别。

```

1 //TOKEN(TK_TYPEDEF, "typedef")
2 //TOKEN(TK_CONST, "const")
3 enum token{
4     TK_BEGIN,
```

```

5 #define TOKEN(k, s) k,
6 #include "token.h"
7 #undef TOKEN
8 };
9 typedef int (*Scanner)(void);
10 static Scanner Scanners[256];
11
12 int GetNextToken(void){
13     int tok;
14
15     PrevCoord = TokenCoord;
16     SkipWhiteSpace();
17     TokenCoord.line = LINE;
18     TokenCoord.col = (int)(CURSOR - LINEHEAD + 1);
19     tok = (*Scanners[*CURSOR])();
20     return tok;
21 }

```

图 2.7 GetNextToken()

在图 2.7 第 19 行，我们用 CURSOR 所指向的当前字符的值来作为数组 Scanners 的下标，该字符的值正好在 0 至 255 之间，而数组 Scanners 是一个由函数指针构成的表格。当然，这个表格 Scanners 当然需要经过初始化，在图 2.2 的 main() 函数中，我们调用 SetupLexer() 来初始化词法分析器，其主要的工作就是初始化这个表格，如图 2.8 所示。

```

1 void SetupLexer(void){
2     int i;
3     // for wchar_t
4     setlocale(LC_CTYPE, "");
5     for (i = 0; i < END_OF_FILE + 1; i++) {
6         if (IsLetter(i)){ // [a-z A-Z _ ]
7             Scanners[i] = ScanIdentifier;
8         }else if(IsDigit(i)){
9             Scanners[i] = ScanNumericLiteral;
10        }else{
11            Scanners[i] = ScanBadChar;
12        }
13    }
14    Scanners[END_OF_FILE] = ScanEOF;
15    Scanners['\''] = ScanCharLiteral;
16    Scanners['"'] = ScanStringLiteral;
17    Scanners['+'] = ScanPlus;
18    Scanners['-'] = ScanMinus;
19    //略
20 }

```

图 2.8 SetupLexer()

该表格共有 256 项，各个以 Scan 为前缀命名的扫描函数的代码并不会太复杂，我们就不一一进行分析，仅以扫描标志符为例，如图 2.9 所示。图 2.9 第 4 至 11 行处理以字母 L 开始的形如 L'a' 的宽字符和形如 L"abc" 的宽字符串。第 12 至 16 行识别标志符，而第 17 行则进一步判断一下识别出来的标志符会不会是 C 语言的关键字，如果确实是 C 程序员声明的标志符（变量名或者函数名等），我们就在第 19 行中记录该标志符的名字。

```

1 static int ScanIdentifier(void){
2     unsigned char *start = CURSOR;
3     int tok;

```

```

4   if (*CURSOR == 'L'){// special case : wide char/string
5     if (CURSOR[1] == '\''){
6       return ScanCharLiteral();           // L'a' wide char
7     }
8     if (CURSOR[1] == '\"'){
9       return ScanStringLiteral(); // L"wide string"
10    }
11  }
12 // letter(letter|digit)*
13 CURSOR++;
14 while (IsLetterOrDigit(*CURSOR)) {
15   CURSOR++;
16 }
17 tok = FindKeyword((char *)start, (int)(CURSOR - start));
18 if (tok == TK_ID){
19   TokenValue.p = InternName((char *)start, (int)(CURSOR - start));
20 }
21 return tok;
22 }

```

图 2.9 ScanIdentifier()

图 2.9 第 19 行的全局变量 TokenValue 记录了当前 token 的值, TokenValue 是个 union value 类型的变量, 联合体 union value 的定义如下所示。UCC 编译器用 union value 来存放单词的值, 而用图 2.7 中的 enum token 来表示单词的类别。

```

union value TokenValue;
union value{
  int i[2];      //存放整数值
  float f;        //存放单精度浮点数
  double d;       //存放双精度浮点数
  void *p;        //指向字符串等较复杂的数据
};

```

## 2.3 UCC 编译器的内存管理

较复杂的 C 或 C++ 程序, 经常会根据其自身应用的特点, 定制其内存堆空间的管理策略, UCC 编译器不也不例外。在第 1 章介绍 `ucc\examples\sc` 时, 为了简单起见, 我们只管使用函数 `malloc()` 来分配堆空间, 并没有去考虑堆空间释放的问题, 这对于 `sc` 这么一个简单的运行时间很短、且堆空间足够用的程序而言, 不会造成什么问题, 在其进程结束时, 操作系统会回收相关内存。但如果长期运行的服务器程序存在内存泄露, 则会造成内存分配不成功, 由此引起致命错误。这也是有些服务器运行一两个月后, 就会突然死机, 重起后又能撑上一两个月的原因之一。在引入请求分页机制的操作系统中, 内存泄露实际上指的是进程的虚地址空间已经完全被分配完了。是的, 物理内存很宝贵, 但内存虚地址空间也是一种宝贵的资源。在请求分页操作系统中, C 程序员面对的地址已经是虚地址, 并不是实际物理内存的地址, 例如, 在以下代码中, “`&abc`” 表示的是变量 `abc` 在进程虚地址空间中的地址。支持请求分页的操作系统, 如 Windows 或 Linux, 已经让程序员跟物理内存说再见了。

```

int abc;
int * ptr = &abc;

```

UCC 对堆空间的需求可分为这么几种情况:

(1) 分析命令行参数、初始化对寄存器的管理和初始化基本的类型系统，如图 2.3 第 4 至 10 行所示。这部分数据在整个编译器运行期间都要存在，在 UCC 中这部分堆空间被称为 ProgramHeap。

(2) 在编译器编译某一个预处理后的 C 文件时，需要为之构建语法树和符号表，一个 C 文件对应一个编译单元。在编译完一个 C 文件后，就可以释放该文件所对应的语法树和符号表。所以这部分空间有着明显的“文件作用域”，在 UCC 中被称为 FileHeap。如图 2.3 的第 11 至 13 行所示。

(3) 在分析各个 C 文件中遇到的字符串及各个标志符的名称，则存放于被称为 StringHeap 的堆空间中。如下所示，在分析名为 abc 的 double 类型变量时，我们会在 StringHeap 中申请堆空间来存放其变量名 abc，之后再遇到名为 abc 的 int 变量，就没有必要再分配内存空间来存放变量名 abc 了。同名的各个标志符完全可以共用同一份字符串空间。

```
double abc = 123.0;
int main(int argc, char * argv[]) {
    int abc = 456;
    return 0;
}
```

这样做的好处是，要判断两个标志符名字是否一样时，不必进行字符串的比较，只要比较标志符对应的字符串首地址是否相等即可。对 UCC 编译器而言，在编译过程中，只有第一次遇到名为 abc 的标志符时，才会为之分配堆空间，之后哪怕是在另外一个 C 文件中，遇到不同类型的标志符 abc，也没有必要再为名为 abc 的标志符分配内存空间了。

对标志符的处理在 uclstr.c 中，如图 2.10 所示。图 2.10 第 5 至 9 行用于在哈希表中寻找是否已经存在同名标志符，如果已经存在，则在第 8 行直接返回。由数据结构的知识，我们知道哈希表可用于快速查找等操作，哈希函数的选择对性能有很大影响。UCC 编译器选择的哈希函数是源于 Unix 系统的 ElfHash() 函数，该函数很适合对字符串的处理。当不存在同名标志符时，第 10 至 18 行会从 StringHeap 中申请堆内存，之后把标志符的名字加入到哈希表 NameBuckets 中。

```
1 char* InternName(char *id, int len) {
2     int i;
3     int h;
4     NameBucket p;
5     h = ELFHash(id, len) & NAME_HASH_MASK;
6     for (p = NameBuckets[h]; p != NULL; p = p->link) {
7         if (len == p->len && strncmp(id, p->name, len) == 0)
8             return p->name;
9     }
10    p = HeapAllocate(&StringHeap, sizeof(*p));
11    p->name = HeapAllocate(&StringHeap, len + 1);
12    for (i = 0; i < len; ++i) {
13        p->name[i] = id[i];
14    }
15    p->name[len] = 0;
16    p->len = len;
17    p->link = NameBuckets[h];
18    NameBuckets[h] = p;
19
20    return p->name;
21 }
```

图 2.10 InternName()

但是，UCC 对 C 程序中字符串的处理则与 GCC 有所不同。例如，以下的代码完全可以

通过编译器的检查，但在 UCC、Clang 和 GCC 上却有着不同的运行结果。

```
#include <stdio.h>
int main(int argc,char * argv[]){
    char * ptr1 = "323";
    char * ptr2 = "323";
    printf("%x %x \n", (unsigned int)ptr1, (unsigned int)ptr2);
    ptr1[0] = '1';
    return 0;
}
```

通过以下实验结果，我们可以发现在运行时，经 `gcc` 和 `clang` 生成的可执行程序 `hello` 会报“段错误（Segmentation fault）”，其原因在于 `gcc` 和 `clang` 把字符串"323"放置在只读的内存中，而语句“`ptr1[0] = '1';`”却企图去改变只读内存。在 PC 机中，这些只读的内存在物理上一般还是存在于 RAM 中，只是在进程页表项上，设置相应物理页面为只读。如果是在嵌入式平台上，这些只读的字符串常量，通常会被连接器放置在 ROM 中，而非 RAM 中。如果 C 编译器把字符串"323"视为只读数据，则不必为相同内容的字符串分配不同的内存。如下所示，我们可以看到，对 `GCC` 编译器而言，上述指针变量 `ptr1` 和 `ptr2` 中的内容都是 8048500，这说明它们指向同一份字符串"323"；对 `Clang` 而言，`ptr1` 和 `ptr2` 中的内容都是 8048510；`UCC` 编译器把字符串当作普通的字符数组来处理，并未置于只读内存中，在内存中有两份内容一样的字符串"323"，且都是可读可写，所以 `ptr1` 的内容为 804a024，而 `ptr2` 的内容为 804a028。而 C89 标准对此的规定是“Identical string literals of either form need not be distinct. If the program attempts to modify a string literal of either form, the behavior is undefined”，请参见 C89 文档 `ansi.c.txt` 的第“3.1.4 String literals”节。由此我们也看到，即便是标准化后的 C 语言，还是有些行为是 C 标准没有明确规定的。

```
iron@ubuntu:demo$ gcc -std=c89 hello.c -o hello;./hello
8048500 8048500
Segmentation fault (core dumped)
iron@ubuntu:demo$ clang -std=c89 hello.c -o hello;./hello
8048510 8048510
Segmentation fault (core dumped)
iron@ubuntu:demo$ ucc hello.c -o hello;./hello
804a024 804a028
iron@ubuntu:demo$
```

字符串的处理函数 `AppendSTR()` 在 `ucl\str.c` 中，主要是进行字符串的复制操作，这里不再啰嗦。前面介绍的 `ProgramHeap`、`FileHeap` 和 `StringHeap` 在 `ucl\ucl.c` 中定义，如图 2.11 第 3 至 6 行所示。第 2 行的指针 `CurrentHeap` 用于指向当前所用的堆空间，该指针指向 `ProgramHeap` 或者 `FileHeap`。在图 2.3 第 12 行调用的函数 `Compile()` 的内部，我们会先调用图 2.11 第 14 行的 `Initialize()` 函数，使指针 `CurrentHeap` 指向 `FileHeap`，之后在第 17 行进行符号表初始化时，所需要的堆内存就从 `FileHeap` 中分配。当完成当前 C 文件的编译后，即 `Compile()` 函数即将返回时，会调用图 2.11 第 21 至 23 行的 `Finalize()` 函数，释放 `FileHeap` 空间，为进行下一个 C 文件的编译做准备。而从 `ProgramHeap` 和 `StringHeap` 中分配的内存，在 UCC 编译器编译完所有 C 文件，即 `main()` 函数结束时，或者中途遇到错误提前退出时，会由操作系统进行回收。

```
1 // CurrentHeap 指向 ProgramHeap 或者 FileHeap
2 Heap CurrentHeap;
3 HEAP(ProgramHeap);
4 HEAP(FileHeap);
5 // all the strings and identifiers
6 HEAP(StringHeap);
```

```

7 // number of warnings in a file
8 int WarningCount;
9 // number of errors in a file
10 int ErrorCount;
11 Vector ExtraWhiteSpace;
12 Vector ExtraKeywords;
13
14 static void Initialize(void){
15 CurrentHeap = &FileHeap;
16 ErrorCount = WarningCount = 0;
17 InitSymbolTable();
18 ASTFile = IRFile = ASMFile = NULL;
19 }
20
21 static void Finalize(void){
22 FreeHeap(&FileHeap);
23 }

```

图 2.11 CurrentHeap

接下来，我们分析一下 UCC 堆空间管理所用到的数据结构和相关函数，具体的代码在 alloc.h 和 alloc.c 中。如图 2.12 所示，第 15 至 20 行的结构体 struct heap 描述了堆空间的内部结构。

```

1 struct mblock{
2     // pointer to next memory block in the heap
3     struct mblock *next;
4     // beginning of memory block
5     char *begin;
6     // currently available memory
7     char *avail;
8     // end of memory block
9     char *end;
10 };
11 union align {
12     double d;
13     int (*f) (void);
14 };
15 typedef struct heap{
16     // pointer to last memory block in the heap
17     struct mblock *last;
18     // memory block list head
19     struct mblock head;
20 } *Heap;
21
22 // In C++, void * --> int * is an error
23 // In C , it is OK for both void * --> int * and int * --> void *
24 // In other words, the type checker of C++ is stricter.
25 #define DO_ALLOC(p) ((p) = HeapAllocate(CurrentHeap, sizeof *(p)))
26 #define ALLOC(p) memset(DO_ALLOC(p), 0, sizeof *(p))
27 #define CALLOC(p) memset(DO_ALLOC(p), 0, sizeof *(p))
28 #define MBLOCK_SIZE (4 * 1024)
29 #define HEAP(hp) struct heap hp = { &hp.head }
30
31 void InitHeap(Heap hp);

```

```

32 void* HeapAllocate(Heap hp, int size);
33 void FreeHeap(Heap hp);

```

图 2.12 struct heap 结构体

由图 2.12 第 29 行的宏定义 HEAP，我们可以发现，在图 2.11 第 3 至 6 行定义的三个对象，实际上都是 struct heap 类型的对象。该对象是由若干个 struct mblock 对象构成的一个单链表，图 2.12 第 19 行的 struct mblock head 充当链表头结点的作用。由数据结构中的知识，我们知道在引入头结点后，就不用再判断链表是否为 NULL。图 2.12 第 17 行的 last 域用于指向这个链表的最末尾的 struct mblock 对象。每个 struct mblock 对象记录了一块通过库函数 malloc() 分配来的内存空间的相关信息，图 2.12 第 5 行的 begin 指向这块内存的起始位置，第 9 行的 end 指向这块内存的结束位置，而第 7 行的 avail 落在区间 [begin, end) 之内，子区间 [begin, avail) 是已经分配出去的内存，而子区间 [avail, end) 则是还未分配出去的内存。图 2.13 描述了以上数据结构。

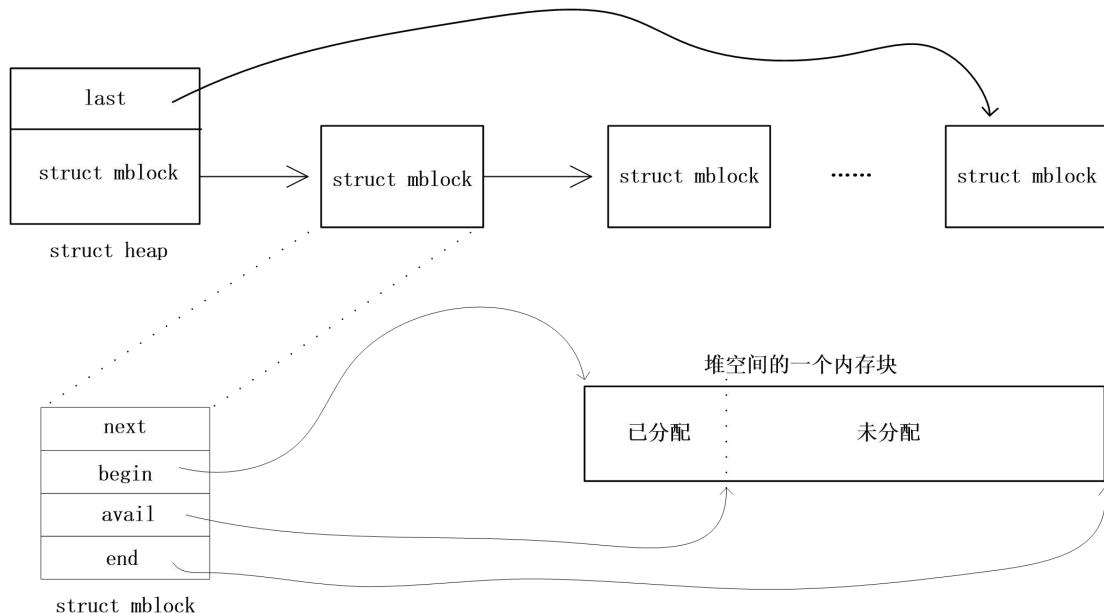


图 2.13 struct mblock 链表

而真正用于堆空间分配的函数 HeapAllocate() 则在 ucl\alloc.c 中，如图 2.14 所示，由图 2.14 第 5 行可知，在 UCC 中，堆空间的分配函数总是试图从 last 所指向的 struct mblock 对象中分配内存。如果子区间 [avail, end) 足够大，则不会执行第 9 行的 while 循环，而是直接执行第 27 行的语句，使指针 avail 右移 size 个字节，而 (avail-size) 即为分配到的堆空间的首地址，之后由第 28 行直接返回。但是，如果 last 所指向的 struct mblock 对象不够大，则要么从空闲块链 FreeBlocks 中找出一块足够大的内存，如第 10 至 12 行所示；要么通过 malloc() 分配一块新的内存空间，如第 14 至 22 行所示。

```

1 void* HeapAllocate(Heap hp, int size){
2     struct mblock *blk = NULL;
3
4     size = ALIGN(size, sizeof(union align));
5     blk = hp->last;
6     /// if the last memory block can't satisfy the request,
7     /// find a big enough memory block from the free block list,
8     /// if there is no such memory block, allocate a new memory block
9     while (size > blk->end - blk->avail) {
10         if ((blk->next = FreeBlocks) != NULL) {
11             FreeBlocks = FreeBlocks->next;
12             blk = blk->next;
13         }
14         else {
15             blk = (struct mblock *)malloc(sizeof(struct mblock));
16             if (blk == NULL)
17                 return NULL;
18             blk->begin = (char *)blk + sizeof(struct mblock);
19             blk->avail = blk->begin;
20             blk->end = blk->avail + size;
21             blk->next = FreeBlocks;
22             FreeBlocks = blk;
23         }
24     }
25     return blk->begin;
26 }

```

```

13     }else{
14         int m = size + MBLOCK_SIZE + sizeof(struct mblock);
15
16         blk->next = (struct mblock *)malloc(m);
17         blk = blk->next;
18         if (blk == NULL){
19             Fatal("Memory exhausted");
20         }
21         blk->end = (char *)blk + m;
22     }
23     blk->avail = blk->begin = (char *) (blk + 1);
24     blk->next = NULL;
25     hp->last = blk;
26 }
27 blk->avail += size;
28 return blk->avail - size;
29 }
```

图 2.14 HeapAllocate()

UCC 中很少直接使用 HeapAllocate() 来分配堆空间，一般都是通过图 2.12 第 26 行的 ALLOC() 和第 27 行的 CALLOC() 宏函数，从 CurrentHeap 所指向的堆空间进行内存分配。而用于堆空间回收的函数 FreeHeap() 如下所示，在 UCC 编译器编译完一个 C 文件时，我们会释放 FileHeap 空间中的内存，在以下堆空间的回收函数 FreeHeap() 中，所做的回收操作只是把 FileHeap 的内存块链表和空闲块链表 FreeBlocks 进行合并，然后通过 InitHeap() 函数重新初始化 FileHeap 对象。实际上，只有 FileHeap 空间会通过 FreeHeap() 函数来释放，而 ProgramHeap 和 StringHeap 对应的堆空间并不会通过 FreeHeap() 来释放，而是在 UCC 编译器结束运行时，由操作系统进行回收。

```

void FreeHeap(Heap hp) {
    hp->last->next = FreeBlocks;
    FreeBlocks = hp->head.next;
    InitHeap(hp);
}
void InitHeap(Heap hp) {
    hp->head.next = NULL;
    hp->head.begin = hp->head.end = hp->head.avail = NULL;
    hp->last = &hp->head;
}
```

## 2.4 C 语言的类型系统

在这一节，我们准备初步讨论一下 C 语言的类型系统，相关的代码主要在 ucltype.c 和 ucltype.h 中。我们知道，一个进程的地址空间可分为代码区和数据区。

对于数据区，C 语言提供了 `char`、`short`、`int`、`long`、`float` 和 `double` 等基本类型来刻画基本的操作数。`char`、`short`、`int` 和 `long` 等整型还进一步分有 `unsigned` 和 `signed`，对大多数编译器而言，缺省时，整型默认为 `signed`。当然也有 C 编译器默认 `char` 为 `unsigned char`。C 标准对基本类型要占多大内存空间，并没有规定得非常死。比如，在一些面向 16 位单片机的 C 编译器中，`int` 就只占 2 个字节；而在 32 位机器上，`int` 一般占 4 个字节。这些基本类型，就如最基本的化学元素一样，按照一定的组合规则，就可以构成更复杂的物质，最终构成了纷繁的大千世界。在 C 语言中，指针、数组和结构体等概念，就相当于是化学元素的组合。

规则，通过这些概念，C 程序员可以描述更加复杂的数据。

C 语言通过引入“函数”的概念，刻画了代码区，对 C 程序员而言，要访问代码区的函数代码时，我们需要知道这段代码的首地址、函数参数和返回值这样的信息，函数名实际上就代表了这段代码的首地址，这些信息可用 C 语言的函数声明来表达，如下所示。

```
1 int f(int,int);
2 int g(int,int);
```

如果忽略掉函数名，则函数 f 和函数 g 拥用相同的特征，同样的参数类型和同样的返回值类型。我们可以说函数 f 和函数 g 的类型是一样的。把“指针”这样的组合规则作用到“函数”上，我们就有了“函数指针（pointer to function）”的概念，由此，我们可以把函数的首地址也当作一种数据来处理。因为整数和浮点数等基本类型相当于基本的化学元素，而指针、结构体、数组和函数等组合规则是在这些基本类型之上衍生出来的，所以我们称这些类型为衍生类型（derived type）。如果把基本类型看成是操作数，把这些组合规则看成是类型运算符，则 C 程序员通过函数或变量的声明，实际上构建了类型表达式，从而告诉 C 编译器我们要如何访问代码区和数据区。

在 UCC 编译器内部，我们需要建立相应的数据结构来刻画基本类型和衍生类型。UCC 编译器是用 C 语言来实现的，很自然的，我们就会用结构体来描述相关类型信息。图 2.15 中的代码来源于 ucl\type.h，为了表述方便，删去了原有的一些注释。

```
1 enum{
2     CHAR, UCHAR, SHORT, USHORT, INT, UINT, LONG, ULONG,
3     LONGLONG, ULONGLONG, ENUM, FLOAT, DOUBLE, LONGDOUBLE,
4     POINTER, VOID, UNION, STRUCT, ARRAY, FUNCTION
5 };
6 // type qualifier
7 enum { CONST = 0x1, VOLATILE = 0x2 };
8
9 #define TYPE_COMMON \
10     int categ : 8; \
11     int qual : 8; \
12     int align : 16; \
13     int size; \
14     struct type *bty;
15
16 typedef struct type{
17     TYPE_COMMON
18 } *Type;
19
20 typedef struct arrayType{
21     TYPE_COMMON
22     int len;           // count of array elements
23 } *ArrayType;
```

图 2.15 struct type

我们通过图 2.15 第 16 行的 struct type 来描述类型信息，而数组类型需要记录更多的信息，我们就用第 20 行的 struct arrayType。可以看到，这两种结构体的开始部分都是 TYPE\_COMMON，我们在第 1 章时介绍过，这相当于 struct arrayType 继承了 struct type。而宏定义 TYPE\_COMMON 如图 2.15 第 9 至 14 行所示，第 10 行的 categ 用来记录类别，第 2 至第 4 行的枚举常量列出了所有的 C 语言类型结构。例如第 2 行的 CHAR 对应的是 char，而 UCHAR 对应的是 unsigned char。第 11 行的 qual 用来记录类型声明时，是否有添加 const 或 volatile 等限定符(qualifier)，qual 的值可以是第 7 行的枚举常量 CONST 或者 VOLATILE。

图 2.15 第 12 行的 align 表示是按多少字节进行对齐的，而第 13 行的 size 则记录该类型要占多少个字节。如果是衍生类型，第 15 行的 bty 用于指向其基类。

我们举个具体的例子来说明。通过 ucl\type.c 中的函数 ArrayOf()、PointerTo() 和 Qualify()，我们可以在基本类型 int 的基础上为“const int”、“int [4]”和“int\*”等类型创建如图 2.16 所示的类型结构。

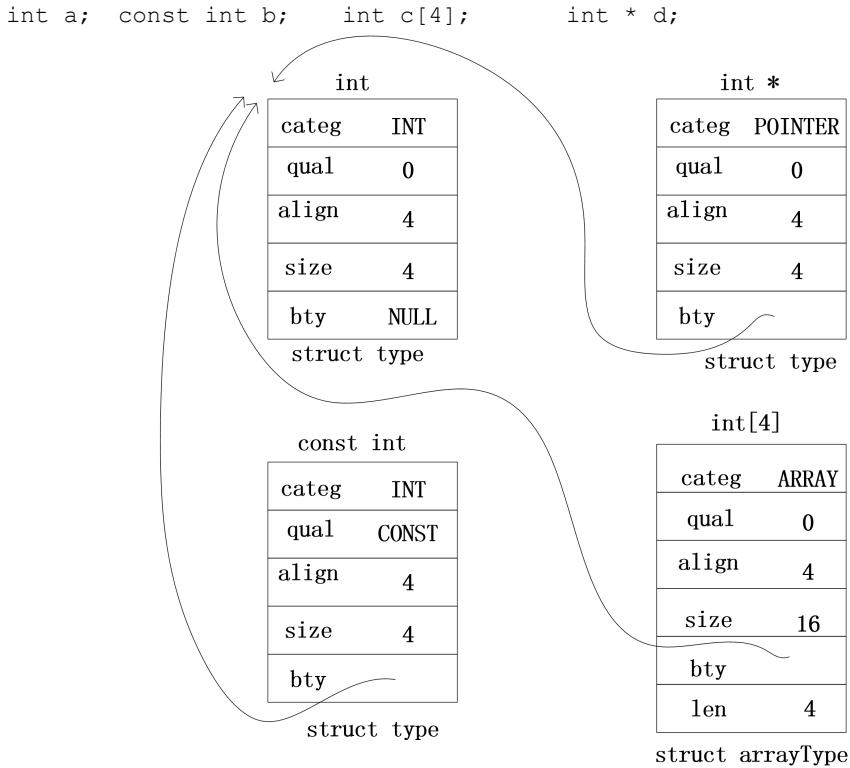


图 2.16 类型结构

由图 2.16，我们可以知道数组 c 的类型由 4 个 int 构成，共占 16 字节，按 4 字节进行对齐，属于 ARRAY 类别；而指针 d 属于 POINTER 类别，占 4 字节。接下来我们再来看一下如何描述结构体类别，如图 2.17 所示。图 2.17 第 11 至 20 行的 struct recordType 用于描述结构体和联合体，第 1 至 9 行的 struct field 用于描述结构体或联合体中的成员域。

```

1  typedef struct field
2  {
3      int offset;
4      char *id;
5      int bits;
6      int pos;
7      Type ty;
8      struct field *next;
9  } *Field;
10
11 typedef struct recordType
12 {
13     TYPE_COMMON
14     char *id;
15     Field flds;
16     Field *tail;
17     int hasConstFld : 16;
18     int hasFlexArray : 16;
19     int complete;
20 } *RecordType;

```

图 2.17 结构体的类型描述

我们仍然结合一个例子来说明。通过 ucl\type.c 的 StartRecord()、AddField() 和 EndRecord() 等函数，我们会为以下结构体 struct Data 构造一个形如图 2.18 的类型结构。

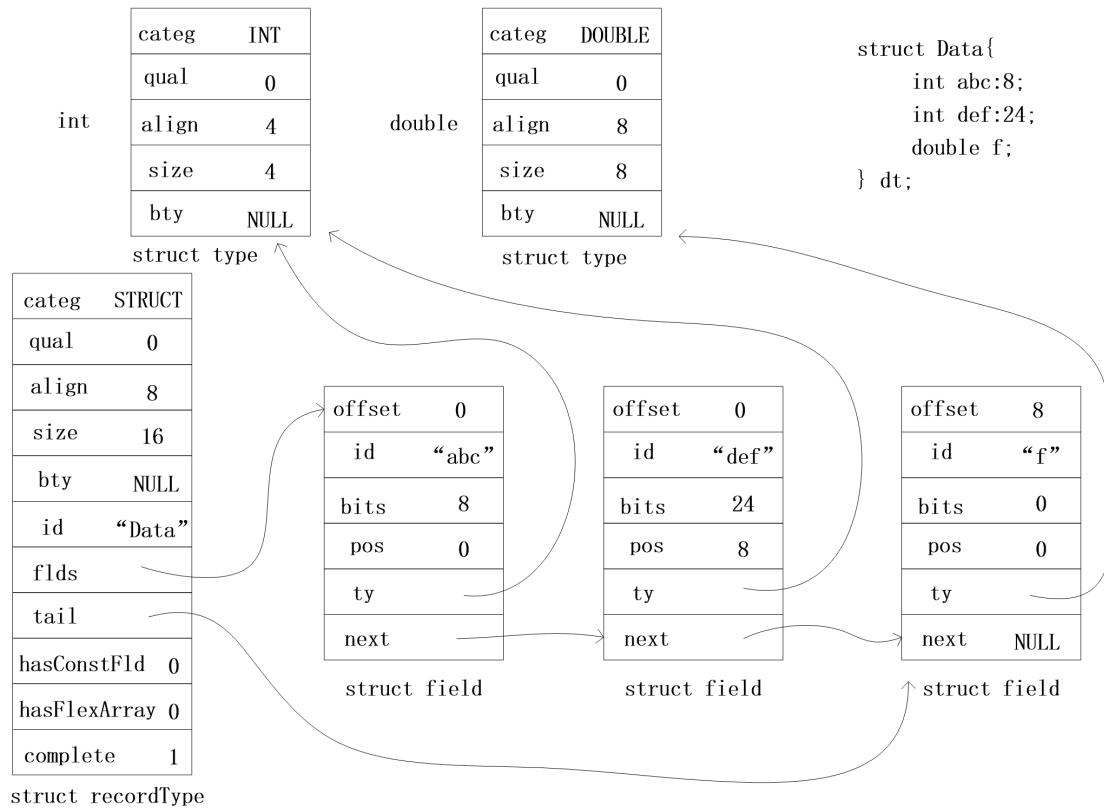


图 2.18 结构体的类型结构

在图 2.18 中，recordType 中记录了结构体 struct Data 类型的大小为 16 字节，其类别为 STRUCT，按 8 字节进行对齐，其中的 flds 指针指向由若干个 struct field 对象构成的链表。每个 struct field 对象描述了结构体中的一个数据域成员，tail 指针相当于指向链尾的 struct field 对象。对于位域（Bit Filed）成员，图 2.17 第 5 行的 bits 记录了其所占用的位数，UCC 会用占 4 字节的整数空间来存放结构体的位域成员。而图 2.17 第 6 行的 pos 则记录了位域在这个整数空间中的起始位置。例如，在图 2.18 的结构体 struct Data 中，成员 abc 和 def 都是位域，abc 占 8 个位，而 def 占 24 个位，它们一共占了 32 位的空间，即 4 字节。UCC 编译器为 struct Data 对象 dt 构造的内存布局如图 2.19 所示。由于 UCC 编译器对 double 类型按 8 字节进行对齐，所以在偏移 offset 为 4 开始的 4 个字节实际上没有放置任何数据。位域 abc 和 def 都处于偏移 0 字节处，但它们的 pos 信息是不一样的；而双精度浮点数 f 位于偏移 8 字节处，占用了 8 字节的内存空间。整个 dt 对象共占 16 字节。不同 C 编译器采取的对齐策略是不一样的，所产生的对象内存布局会有所不同。

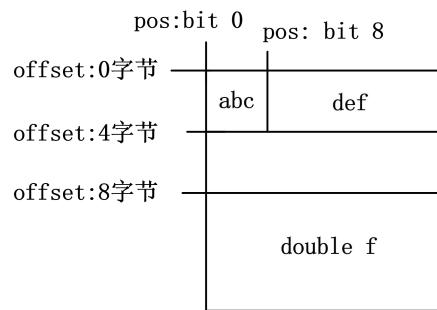


图 2.19 struct Data 对象 dt 的内存布局

接下来，我们来看一下 UCC 的类型系统是如何描述“函数”的，如图 2.20 所示。图 2.20 的第 15 至 19 行的 struct functionType 描述了与函数相关的类型信息，TYPE\_COMMON

中的 bty 记录了函数返回值的类型信息，而第 18 行的 sig 则记录了参数列表的类型信息。而图 2.20 第 1 至 6 行的 struct parameter 则用于描述函数的某个形参的类型信息。函数的各个参数构成一个向量，保存于图 2.20 第 12 行的 params 域中。

```

1  typedef struct parameter
2  {
3      char *id;
4      Type ty;
5      int reg;
6  } *Parameter;
7  //用于描述函数的参数列表
8  typedef struct signature
9  {
10     int hasProto : 16;
11     int hasEllipsis : 16;
12     Vector params;
13 } *Signature;
14 //用于描述函数的参数列表和返回值等信息
15 typedef struct functionType
16 {
17     TYPE_COMMON
18     Signature sig;
19 } *FunctionType;

```

图 2.20 函数的类型描述

在最古老的 C 语言中，函数的参数列表并不是函数接口的一部分，之后随着 C 语言的发展，对函数的参数进行了更严格的类型检查，函数的参数列表也就成了函数接口的一部分。因此，C 语言函数可分为旧式风格 (old-style) 或者新式风格 (new-style) 的函数。如图 2.21 第 3 至 9 行的函数 f1 和函数 f2 就属于旧式风格，而第 10 至 15 行的函数 f3 和函数 f4 就属于新式风格。图 2.21 的第 17 至 22 行的函数调用都是合法的，由此可知，C 编译器对旧式风格的函数甚至连实参的个数都不进行检查。与之形成鲜明对比的是，第 24 至 27 行注释中的代码都无法通过编译器的检查，由此可知，C 编译器对新式风格的函数会进行更严格类型检查。这也是图 2.20 第 10 行的 hasProto 的含义，hasProto 是“存在原型 (has prototype)”的意思，即函数的调用者要严格按照函数声明时的样子来调用之，参数列表已成为函数接口的一部分。作为 C 程序员，应尽量不去使用旧式风格的函数定义或声明。毕竟，由于历史上使用旧式风格的函数引起了不少问题，我们才会引入类型检查更严格的新式风格函数。但是作为 C 编译器，却需要背上这个历史的包袱，新旧风格都需要去支持。

```

1  #include <stdio.h>
2
3  int f1(){
4      return 3;
5  }
6  int f2(a,b)
7  int a,b;
8  return a+b;
9 }
10 int f3(void){
11     return 3;
12 }
13 int f4(int a,int b){
14     return a+b;
15 }
16 int main(int argc,char * argv[]){
17     f1();
18     f1(3);
19     f1(3,2);
20     f2();
21     f2(3);
22     f2(3,2);
23     f3();
24     //f3(3);
25     //f3(3,2);
26     //f4();
27     //f4(3);
28     f4(3,2);
29     return 0;
30 }

```

图 2.21 旧式风格和新式风格的函数

而图 2.20 第 11 行的 hasEllipsis 则用于判断新式风格的函数中是否存在变参，ellipsis 是省略号的意思，在 C 语言中 printf 就是一个最典型的变参函数，其函数接口如下所示。

```
int printf(const char *format, ...);
```

接下来我们以如下所示的函数 f5 为例来说明函数对应的类型结构。通过 ucl\type.c 中的 FunctionReturn() 等函数，我们为之构造一个如图 2.22 所示的类型结构。

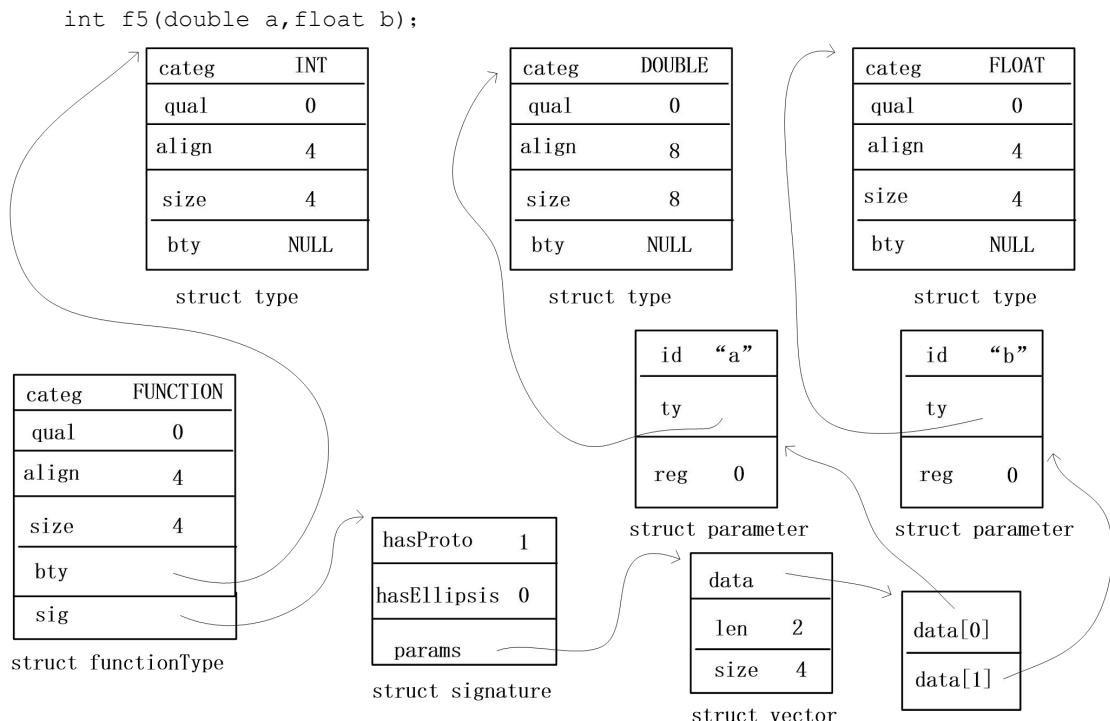


图 2.22 函数的类型结构

在图 2.22 中，结构体 `struct parameter` 描述了函数的某个参数的相关信息，`id` 为形参的名字，`ty` 为形参的类型，而 `reg` 表示形参声明时是否有 `register` 这样的说明符，该说明符只是建议 C 编译器把形参尽量放到寄存器中。而 `struct signature` 则描述了参数列表的相关信息，`hasProto` 为 1 时，表示是新式风格的函数，此时 `params` 域指向一个向量，该向量包含多个 `struct parameter` 对象。

通过这一节的图 2.16、图 2.18 和图 2.22，我们对 UCC 编译器是如何刻画 C 语言的数组、结构体和函数等类型信息会有一个非常直观的感觉。UCC 编译器会在语法分析和语义检查时进行这些类型结构的构建，我们会在后续章节再进行讨论。而 `int` 和 `double` 等基本类型的结构，则通过调用函数 `SetupTypeSystem()` 来构建，如图 2.23 所示。图 2.23 第 5 至 21 行创建了基本类型和指针类型，指定了这些类型的大小 (`size`)、对齐 (`align`) 和类别 (`categ`) 等信息。图 2.23 第 26 至 38 行创建了一个形如“`int f();`”的缺省函数类型 `DefaultFunctionType`，这是一个旧式风格的函数。

```

1 void SetupTypeSystem(void) {
2     int i;
3     FunctionType fty;
4
5     T(CHAR)->size = T(UCHAR)->size = CHAR_SIZE;
6     T(SHORT)->size = T(USHORT)->size = SHORT_SIZE;
7     T(INT)->size = T(UINT)->size = INT_SIZE;
8     T(LONG)->size = T(ULONG)->size = LONG_SIZE;
9     T(LONGLONG)->size = T(ULLONG)->size = LONG_LONG_SIZE;

```

```

10 T(FLOAT)->size = FLOAT_SIZE;
11 T(DOUBLE)->size = DOUBLE_SIZE;
12 T(LONGDOUBLE)->size = LONG_DOUBLE_SIZE;
13 T(POINTER)->size = INT_SIZE;
14 // without this, TypeToString() would have Segmentation fault
15 T(POINTER)->bty = T(INT);
16
17 // type category, type alignment
18 for (i = CHAR; i <= VOID; ++i){
19     T(i)->categ = i;
20     T(i)->align = T(i)->size;
21 }
22 *****
23     Construct a default function type:
24     int f();
25 *****
26 ALLOC(fty);
27 fty->categ = FUNCTION;
28 fty->qual = 0;
29 fty->align = fty->size = T(POINTER)->size;
30 // the type of the return value
31 fty->bty = T(INT);
32 // no prototype, old-style declaration.
33 ALLOC(fty->sig);
34 CALLOC(fty->sig->params);
35 fty->sig->hasProto = 0;
36 fty->sig->hasEllipsis = 0;
37
38 DefaultFunctionType = (Type)fty;
39 WCharType = T(WCHAR);
40 }

```

图 2.23 SetupTypeSystem()

在 C 语言中，如果有一个函数未经声明就直接使用，则 C 编译器会把这个函数当作旧式风格的函数，用图 2.23 第 38 行的 DefaultFunctionType 做为该函数的类型，不对该函数的参数进行任何的检查。我们举一个例子来说明这会引起多让人莫名其妙的问题。

假设有两个 C 文件，一个文件名为 b.c，其中定义了一个函数 fadd，用于对两个 float 类型的浮点数进行加法运算；而另一个文件名为 a.c，其中调用了 fadd(3.0f,3.0f)。这个程序非常简单，只要学过几天 C 语言的人几乎都会预期这个程序输出的结果是 6.0。

```

//a.c
#include <stdio.h>
int main(int argc,char * argv[]){
    fadd(3.0f,3.0f);
    return 0;
}
//b.c
#include <stdio.h>
void fadd(float a,float b){
    float c;
    c = a+b;
    printf("%f\n",c);
}

```

但上机运行后的结果竟然是 2.125，这一定会让我们大吃一惊。其运行结果如下所示。

我们看到 Clang 至少给了我们一点警告提示, 让我们知道原来在 a.c 中调用的函数 fadd 是“没有声明就直接使用”。这就导致在 a.c 中, 函数 fadd 被当作旧式风格的函数。C 编译器视 fadd 的类型为图 2.23 中的 DefaultFunctionType。

```
iron@ubuntu:~/src/ucc162_3/demo$ ucc a.c b.c -o ab;./ab
(exprchk.c,314):(a.c,4):warning:Too many arguments
2.125000
iron@ubuntu:~/src/ucc162_3/demo$ gcc a.c b.c -o ab;./ab
2.125000
iron@ubuntu:~/src/ucc162_3/demo$ clang a.c b.c -o ab;./ab
a.c:4:2: warning: implicit declaration of function 'fadd' is invalid in C99
      [-Wimplicit-function-declaration]
      fadd(3.0f,3.0f);
      ^
1 warning generated.
2.125000
```

这就是噩梦的源头。只要在 a.c 中加上声明 “void fadd(float a,float b);” 后, 再调用 fadd() 函数, 我们就得到了预期的结果 6.0。下面, 我们就来分析一下, 为什么旧式风格的函数会带到这么怪异的问题。为了说明方便, 我们把上述程序稍微修改一下, 适当加了一些输出语句, 如图 2.24 所示。

```
1 // a.c
2 #include <stdio.h>
3 //void fadd(float a,float b);
4 int main(int argc,char * argv[]){
5     float f = 3.0f;
6     float f2 = 2.125f;
7     double d = 3.0;
8     printf("f = %x\n",*((int *)&f));
9     printf("f2 = %x\n",*((int *)&f2));
10    printf("d = %x %x\n",*((int *)&d),*((int *)&d)+1));
11
12    fadd(3.0f,3.0f);
13    return 0;
14 }
15
16 // b.c
17 #include <stdio.h>
18 void fadd(float a,float b){
19     float c;
20     c = a+b;
21     printf("a = %x \n",*(int*)(&a));
22     printf("b = %x \n",*(int*)(&b));
23     printf("c = %x \n",*(int*)(&c));
24     printf("%f\n",c);
25 }
```

图 2.24 被误当作旧式风格的函数

再次上机运行, 我们会得到如下所示结果。按照 IEEE754 的浮点数编码格式, float 类型的浮点数 3.0f 在内存中对应的十六进制数值为 0x40400000, 而 2.125f 则对应 0x40080000, double 类型的 3.0 在内存中对应 8 个字节, 内容为 [0x00000000 0x40080000]。同时, 我们发现, 在 b.c 中, 形参 a 的值竟然是 0x00000000, 形参 b 的值竟然是 0x40080000, 这就相当于是 2.125f 和 0.0f 相加, 结果当然是 2.125f。

```
iron@ubuntu:~/src/ucc162_3/demo$ ucc a.c b.c -o ab;./ab
```

```
(exprchk.c,314):(a.c,12):warning:Too many arguments
f = 40400000
f2 = 40080000
d = 0 40080000
a = 0
b = 40080000
c = 40080000
2.125000
iron@ubuntu:~/src/ucc162_3/demo$ gcc a.c b.c -o ab;./ab
f = 40400000
f2 = 40080000
d = 0 40080000
a = 0
b = 40080000
c = 40080000
2.125000
```

由于 fadd 函数未经声明就使用，因此 C 编译器在 a.c 中把 fadd 函数当作旧式风格，按 IT 大佬们的约定，C 编译器对旧式风格的函数会进行一个被称为“实参提升”的动作。我们在第 1.7 节讨论 C 语言变参函数时介绍过，C 编译器对变参函数中的无名参数也会进行实参提升。如 ucltype.c 中的 Promote() 函数所示，凡是低于 int 型的其他整型，包括 char 和 short 都会被提升为 int，而单精度 float 则会被提升为 double；其他类型保持不变。

```
Type Promote(Type ty) {
    return ty->categ < INT ? T(INT) : (ty->categ == FLOAT ? T(DOUBLE) : ty);
}
```

于是，C 编译器面对未声明就使用的函数调用“fadd(3.0f,3.0f);”时，默默地进行了实参提升的操作。真正执行的函数调用是“fadd(3.0,3.0);”，压入栈的是两个 double 类型的浮点数 3.0，共占了 16 字节。在小端机器上，浮点数 3.0 的存放如图 2.25 所示，在低地址的 4 字节中存放 0x00000000，在高地址的 4 字节中存放 0x40080000。但在文件 b.c 中，函数 fadd 是新式风格的函数，其接口为“void fadd(float a,float b)”，按照函数声明，仍然把形参 a 和 b 当作 float 来处理，依照 C 调用约定，参数从右向左入栈，所以形参 a 对应的是 0x00000000，形参 b 对应的是 0x40080000。

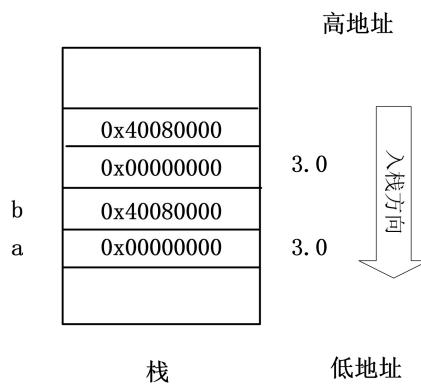


图 2.25 栈示意图

总之，远离旧式风格的 C 函数，同时记住，函数要先声明再使用，否则我们就不自觉地在使用旧式风格的函数声明。上述例子阐述了旧式风格的函数所带来的噩梦。

## 2.5 UCC 编译器的符号表管理

这一节，我们准备初步讨论一下 UCC 编译器的符号表管理，与符号表管理相关的代码主要在 ucl\symbol.h 和 ucl\symbol.c 中。UCC 编译器内部需要对用到的所有符号进行分类，并建立相应的数据结构来记录与符号相关的信息。图 2.26 第 25 行的结构体 struct symbol 用于记录与符号相关的信息，这些信息如图第 7 至 23 行所示。第 8 行的 kind 用于区别不同类别的符号，其取值范围如图第 1 至 5 行所示；第 9 行的 name 用于记录符号的名称；第 11 行的 ty 用于记录符号的类型；第 23 行的 pcoord 用于记录符号所在的文件位置，在调试时方便定位。其余各行的作用我们以后会结合上下文进行讨论。

```

1 enum {
2     SK_Tag,      SK_TypeInfo,   SK_EnumConstant, SK_Constant,
3     SK_Variable, SK_Temp,    SK_Offset,   SK_String,
4     SK_Label,     SK_Function, SK_Register
5 };
6
7 #define SYMBOL_COMMON \
8     int kind;           \
9     char *name;         \
10    char *aname;        \
11    Type ty;            \
12    int level;          \
13    int sclass;         \
14    int ref;             \
15    int defined : 1;    \
16    int addressed : 1;  \
17    int needwb : 1;     \
18    int unused : 29;    \
19    union value val;   \
20    struct symbol *reg; \
21    struct symbol *link; \
22    struct symbol *next; \
23    Coord pcoord;
24
25 typedef struct symbol
26 {
27     SYMBOL_COMMON
28 } *Symbol;

```

图 2.26 struct symbol

下面，我们结合一个具体的 C 程序来熟悉一下 UCC 编译器内部对符号所做的分类，如图 2.27 所示。

```

1 struct Data{                                14 d = a + b;
2     int num1;                                15 a = 3;
3     int num2;                                16 dt.num2 = a + b;
4 } dt;                                         17 return 0;
5 enum Color{                                 18 }
6     RED, GREEN, BLUE                         19
7 };                                         20 function main
8 typedef int INT32;                           21 t0 :&str0;
9 int a,b,c,d;                               22 hi = t0;
10 int main(int argc,char * argv[]){          23 t1 : a + b;
11     char * hi = "Hello World.";              24 c = t1;
12     Begin:                                    25 d = t1;
13     c = a + b;                             26 a = 3;

```

```

27 t3 : a + b;
28 dt[4] = t3;
29 return 0;
30 ret

```

图 2.27 各种不同类别的符号

图 2.27 的第 1 至 18 行是一个简单的 C 程序，而第 20 至 30 行是由 UCC 编译器产生的中间代码。结合图 2.26 和图 2.27，我们能看到以下几种不同的符号：

- (1) 结构体名 Data 和枚举名 Color，对应的类别为 SK\_Tag;
- (2) 由关键字 typedef 给已有类型 int 取的别名 INT32，对应的类别为 SK\_TypeInfo;
- (3) 枚举常量 RED、GREEN 和 BLUE，对应的类别为 SK\_EnumConstant;
- (4) 常量 3，对应的类别为 SK\_Constant;
- (5) 全局变量名 dt、a、b、c、d 和局部变量 hi，对应的类别为 SK\_Variable;
- (6) 临时变量 t0、t1 和 t2，对应的类别为 SK\_Temp;
- (7) 用于访问结构体成员的 dt.num2，经 UCC 编译器处理后，我们更关注域成员 num2 在结构体 Data 中的偏移，因为结合 dt 的首地址和偏移 4，我们就能定位 dt.num2 的具体位置。在 UCC 编译器内部，为 dt.num2 取的名字为 dt[4]，对应的类别为 SK\_Offset;
- (8) 第 21 行的 str0，是 UCC 编译器内部为第 11 行的字符串"Hello World."所取的名字，对应的类别为 SK\_String;
- (9) 标号名 Begin，对应的类别为 SK\_Label;
- (10) 函数名 main，对应的类别为 SK\_Function;

UCC 编译器把汇编代码中出现的形如 "%eax" 这样的寄存器名也当作符号来管理，对应的类别为 SK\_Register。这 11 种符号正是图 2.26 第 2 和第 4 行所列出的各种不同类别的符号。实际上，当寄存器 eax 中存放的是地址，而非一般数值时，我们是把寄存器当指针变量来使用，在汇编代码中表示为“(%eax)”，在 UCC 内部会对应一个特殊的类别 SK\_IRegister，是 Indirect Register 的缩写，代表通过寄存器进行间接寻址。所以，UCC 编译器要管理的符号共有 12 种。这 12 种符号中，有些符号的相关信息用 struct symbol 对象就可以记录；而有些符号需要记录更多的信息，比如 SK\_Variable、SK\_Temp 和 SK\_Offset 类别的符号，就需要用到图 2.28 第 17 至 24 行所示的 struct variableSymbol。

```

1 typedef struct valueDef
2 {
3     Symbol dst;
4     int op;
5     Symbol src1;
6     Symbol src2;
7     BBlock ownBB;
8     struct valueDef *link;
9 } *ValueDef;
10 // the use of a definition
11 typedef struct valueUse
12 {
13     ValueDef def;
14     struct valueUse *next;
15 } *ValueUse;
16
17 typedef struct variableSymbol
18 {
19     SYMBOL_COMMON
20     InitData idata;
21     ValueDef def;
22     ValueUse uses;

```

```

23 int offset;
24 } *VariableSymbol;

```

图 2.28 variableSymbol

在图 2.28 的第 19 行，我们再次看到 SYMBOL\_COMMON，这说明我们可以把 struct variableSymbol 当成是 struct symbol 的子类。图 2.28 第 23 行的 offset 用于记录 SK\_Offset 类别的符号的偏移值，例如上述的 dt[4] 中的 4；第 20 行的 idata 记录用于变量初始化的值。接下来，我们来说明一下图 2.28 第 21 行的 def 和第 22 行的 uses 的作用。为了突出“公共子表达式”的概念和表述上的方便，我们从图 2.27 中抽取关键的几行代码及其对应的中间代码，整理为图 2.29。

1 // C 代码	7 t1 : a + b;
2 c = a + b;	8 c = t1;
3 d = a + b;	9 d = t1;
4 a = 3;	10 a = 3;
5 dt.num2 = a + b;	11 t3 : a + b;
6 //中间代码	12 dt[4] = t3;

图 2.29 公共子表达式

在图 2.29 中，我们可以发现第 2 行的子表达式 “a+b” 确实进行了计算，其结果存到第 7 行的临时变量 t1 中，但是第 3 行的 “a+b” 则没有必要再次计算，因为此时 a 和 b 都没有发生变化，我们可以沿用临时变量 t1 中的保存的值。但是第 4 行对 a 进行了修改，导致在 t1 中保存的 “a+b” 失效，此时第 5 行的 “a+b” 就需要重新进行计算。UCC 编译器期望其产生的汇编代码能减少不必要的计算，由此引入了图 2.28 第 1 至 9 行的 struct valueDef，即“value definition”的缩写。当我们进行 “a+b”的计算后，就产生了一个新的 value，我们用 valueDef 结构体中的 src1 来记录操作数 a，用 src2 来记录操作数 b，用 op 来记录运算符 ‘+’，而用 dst 来记录运算结果 t1，图 2.28 第 7 行的 ownBB 则用来记录这个 value 是在哪个基本块中产生的。在中间代码生成阶段，UCC 编译器会把生成的中间代码划分在不同的基本块中，每个基本块相当于一个原子，其中的代码要么都不执行，要么都执行。如果要在不同基本块之间重用已经计算出来的形如 “a+b”的值，则需要进行较复杂的数据流分析。UCC 编译器为了简单起见，仅在同一个基本块内，对形如 “a+b” 这样的公共子表达式进行重用。下面，我们以图 2.29 第 7 行的 “t1 : a + b;” 为例，说明由图 2.28 第 1 至 15 行的 valueDef 和 valueUse 所形成的数据结构，如图 2.30 所示。

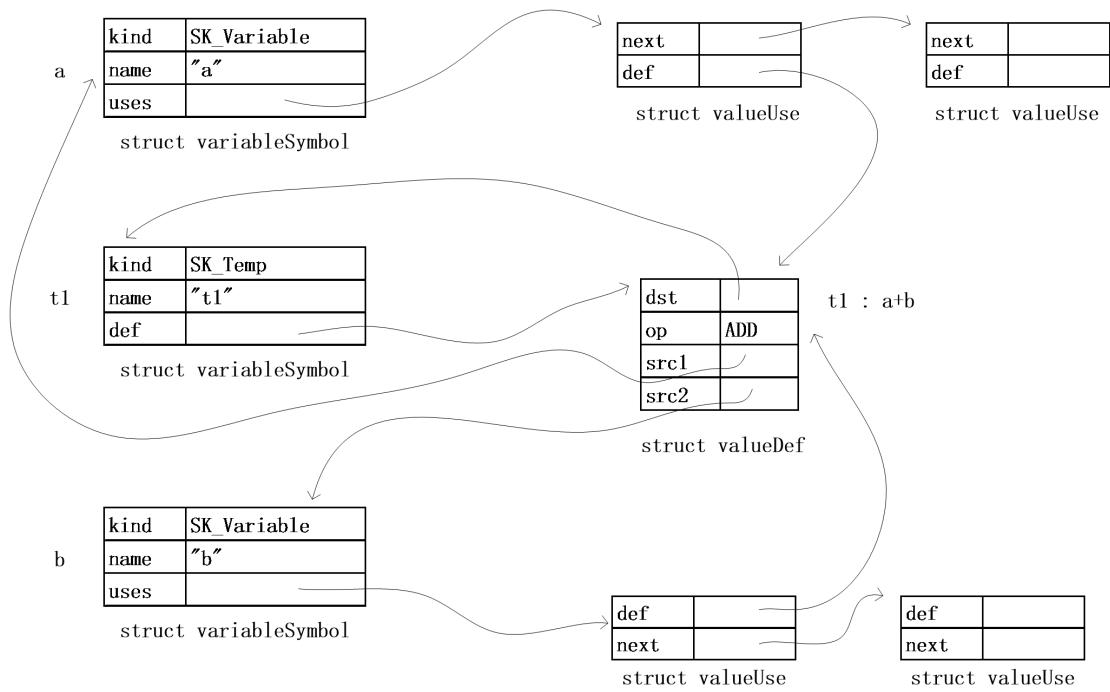


图 2.30 valueDef 和 valueUse

从图 2.30，我们可以看到，变量 *a* 对应的 *uses* 域实际上是由若干个 *struct valueUse* 对象构成的一个链表。每个 *struct valueUse* 对象的 *def* 域记录了变量 *a* 在哪个子表达式中被使用。当图 2.29 第 4 行给 *a* 重新赋值后，我们可以使 *a* 的 *uses* 链上的每个 *struct valueUse* 对象记录的 *valueDef* 失效。这是 *uclgen.c* 中的函数 *TrackValueChange()* 所要完成的功能，如图 2.31 所示。图 2.31 第 14 行的代码使 *valueDef* 对象中的 *dst* 域为 *NULL*，这就使得公共子表达式 “*a+b*” 不再有效。当然不论变量 *a* 的内容如何变化，在变量 *a* 的生命周期内，其地址是不会变化的，所以子表达式 “*&a*” 会一直有效，这正是图 2.31 第 13 行的 if 条件所要检测的情况，即当 *valueDef* 对象中的 *op* 域不是取地址运算时，才执行第 14 行从而使 *struct valueDef* 对象失效。

```

1 static void TrackValueChange(Symbol p) {
2     ValueUse use = AsVar(p)->uses;
3
4     assert(p->kind == SK_Variable);
5     while (use) {
6         ****
7             mark the definitions using @p as operand invalid.
8             Unless it contains the address of a variable.
9             The content of a variable changes while its address not.
10            So the temporary is still valid in this case.
11            t1 = &var; //var may change, but &var not.
12            ****
13            if (use->def->op != ADDR) {
14                use->def->dst = NULL;
15            }
16            use = use->next;
17    }
18 }
```

图 2.31 TrackValueChange()

从图 2.30 可以发现，到目前为止，我们还没有解决 *struct symbol* 对象 *a*、*b* 和 *t1* 如何存

放的问题，及如何快速找到 valueDef 对象的问题。这需要我们引入图 2.32 的两个结构体。

```

1  typedef struct functionSymbol{
2      SYMBOL_COMMON
3      Symbol params;
4      Symbol locals;
5      Symbol *lastv;
6      int nbblock;
7      BBlock entryBB;
8      BBlock exitBB;
9      ValueDef valNumTable[16];
10 } *FunctionSymbol;
11
12 typedef struct table{
13     Symbol *buckets;
14     int level;
15     struct table *outer;
16 } *Table;

```

图 2.32 struct table

图 2.32 第 1 至 10 行的 struct functionSymbol，用来描述 SK\_Function 类别的符号（函数名）的相关信息，第 3 行的 params 记录了函数形参所构成的单向链表，而第 4 行的 locals 用于记录局部变量构成的单向链表，第 6 行的 nbblock 代表函数体中的基本块个数，第 7 行的 entryBB 是入口的基本块，而 exitBB 是出口的基本块。而第 9 行的 valNumTable[16] 则是一个哈希表，用于快速查找形如“a+b”的公共子表达式，如图 2.33 所示。图中标为 struct symbol 的对象可能是 struct symbol 对象，也可能是其“子类”的对象，例如 struct variableSymbol 对象。

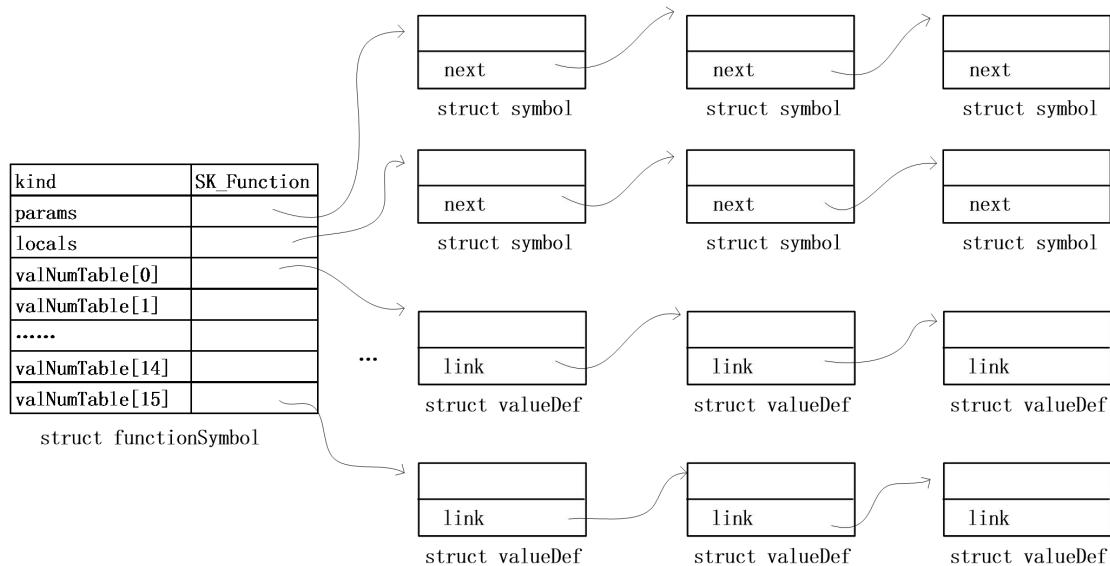


图 2.33 valueNumTable 哈希表

为了查找哈希表，我们还需要定义相应的哈希函数来计算哈希值。图 2.34 第 4 行就是一个简单的哈希函数，由于哈希表 valueNumTable 中只定义了 16 个链表，所以整数 h 的值要落在区间 [0,15] 之间。如果已在 C 程序中使用过全局变量 a 的地址，例如使用过“ptr = &a;”，则之后既可以通过“\*pt=3;”也可以通过“a = 3;”来改变 a 的值，此时要判断变量 a 的内容是否发生变化就需要做更复杂的分析。UCC 编译器采取的策略很简单，此时就不再重用“a+b”这样的已计算过的值，而是重新进行计算。图 2.34 第 9 行的 if 语句就是针对这种情况进行判断。

```

1
2 Symbol TryAddValue(Type ty, int op, Symbol src1, Symbol src2)
3 {
4     int h = ((unsigned)src1 + (unsigned)src2 + op) & 15;
5     ValueDef def = FSYM->valNumTable[h];
6     Symbol t;
7     assert(op != CVTI4F4 && op != CALL && op != MOV);
8
9     if (op != ADDR && (src1->addressed || (src2 && src2->addressed)))
10    goto new_temp;
11
12    while (def) {
13        if (def->op == op && (def->src1 == src1 && def->src2 == src2))
14            break;
15        def = def->link;
16    }
17
18    if (def && def->ownBB == CurrentBB && def->dst != NULL)
19        return def->dst;
20
21 new_temp:
22    t = CreateTemp(ty);
23    GenerateAssign(ty, t, op, src1, src2);
24    def = AsVar(t)->def;
25    def->link = FSYM->valNumTable[h];
26    FSYM->valNumTable[h] = def;
27    return t;
28 }

```

图 2.34 TryAddValue()

图 2.34 第 12 至 19 行用于在哈希表中查找是否存在已计算过的有效表达式。在第 18 行我们看到了对 `def->dst` 是否为 `NULL` 的判断，而在图 2.31 的 `TrackValueChange()` 函数中，我们正是通过 “`use->def->dst = NULL;`” 来使表达式 “`a+b`” 无效。如果能找到，则从图 2.34 第 19 行返回，否则我们需要重新计算表达式，并把结果保存到第 22 行创建的临时变量中，之后还要在第 24 至 26 行把新生成的 `valueDef` 对象加入到哈希表 `valNumTable` 中。

以上我们解决了如何快速查找形如 “`a+b`” 的公共子表达式的问题。另一个问题就是大量的 `struct symbol` 对象要如何存放的问题。图 2.32 中的第 12 至 16 行的 `struct table` 正是用来做这个事情的。这也是我们本节标题中所说的符号表。查看 `ucl\symbol.c` 的函数 `AddSymbol()`，我们不难发现，符号表仍然是采用 “哈希表”的数据结构，如图 2.35 所示。

```

1 #define SYM_HASH_MASK 127
2 //允许同一个 symbol 对象出现在多个符号表中
3 typedef struct bucketLinker{
4     struct bucketLinker * link;
5     Symbol sym;
6 } * BucketLinker;
7
8 static Symbol AddSymbol(Table tbl, Symbol sym){
9     unsigned int h = (unsigned long)sym->name & SYM_HASH_MASK;
10    BucketLinker linker;
11    CALLOC(linker);
12    if (tbl->buckets == NULL) {
13        int size = sizeof(Symbol) * (SYM_HASH_MASK + 1);

```

```

14     tbl->buckets = HeapAllocate(CurrentHeap, size);
15     memset(tbl->buckets, 0, size);
16 }
17 // add the new symbol into the first positon of bucket[h] list.
18 linker->link = (BucketLinker)tbl->buckets[h];
19 linker->sym = sym;
20 sym->level = tbl->level;
21 tbl->buckets[h] = (Symbol)linker;
22
23 return sym;
24 }
25

```

图 2.35 AddSymbol()

由于同一个函数名或变量名可以在不同作用域中被多次声明，我们需要把同一个符号加入到不同的符号表中，为此，我们在图 2.35 第 3 至 6 行引入结构体 bucketLinker，每当一个符号 sym 被加入到某个符号表 tbl 时，我们就会在图 2.35 第 11 行为之创建一个 bucketLinker 对象。当一个符号出现在多个符号表时，就会对应多个 bucketLinker 对象，通过这些 bucketLinker 对象，我们可找到对应的符号。符号表实际上就是一个哈希表，所采用的哈希函数在图 2.35 第 9 行，宏 SYM\_HASH\_MASK 的值为 127。当哈希表为空时，我们会通过图 2.35 的第 12 至 16 行，从堆中分配 128 个哈希桶。第 17 至 21 行用于往符号表里添加一个新的符号。当然，我们往哈希桶中添加的是一个 bucketLinker 对象，该对象的 sym 域指向了对应的符号。

下面，我们再结合一个简单的 C 程序，说明一下图 2.32 第 15 行 outer 成员的作用。

```

1 // 深度 level is 0
2 int a = 10;
3 int main(int argc,char * argv[]){    // level is 1
4     { // level is 2
5         int a = 20;
6         int b[10] = {1,2};
7         { // level is 3
8             int a = 30;
9             a = 40;
10        }
11        a = 50;
12    }
13    a = 60;
14    return 0;
15 }
16

```

图 2.36 C 语言的作用域

如图 2.36 所示，第 2 行定义了一个全局变量 a，第 5 行定义了一个同名的局部变量 a，而在第 8 行又定义了一个局部变量 a。在 C 语言中，一对大括号一般代表一个新的作用域。当然，第 6 行的大括号是用于初始化数组，并不代表一个作用域。由 C 的文法，复合语句以左大括号开始，之后跟着若干个声明，再跟上由若干条语句，最后是右大括号。在 C 语言中，函数体实际上就是一个复合语句，如图 2.36 的第 3 至第 15 行所示。当然，复合语句内部还可以包含新的复合语句。

compound-statement:

```
{ declaration-listopt statement-listopt }
```

每一个复合语句对应一个新的作用域，在 UCC 编译器内部，每当进入一个新的作用

域时，我们就会创建一张新的符号表，用来记录在该作用域中声明的符号。图 2.37 给出了 UCC 编译器为图 2.36 的 C 程序所创建的符号表。

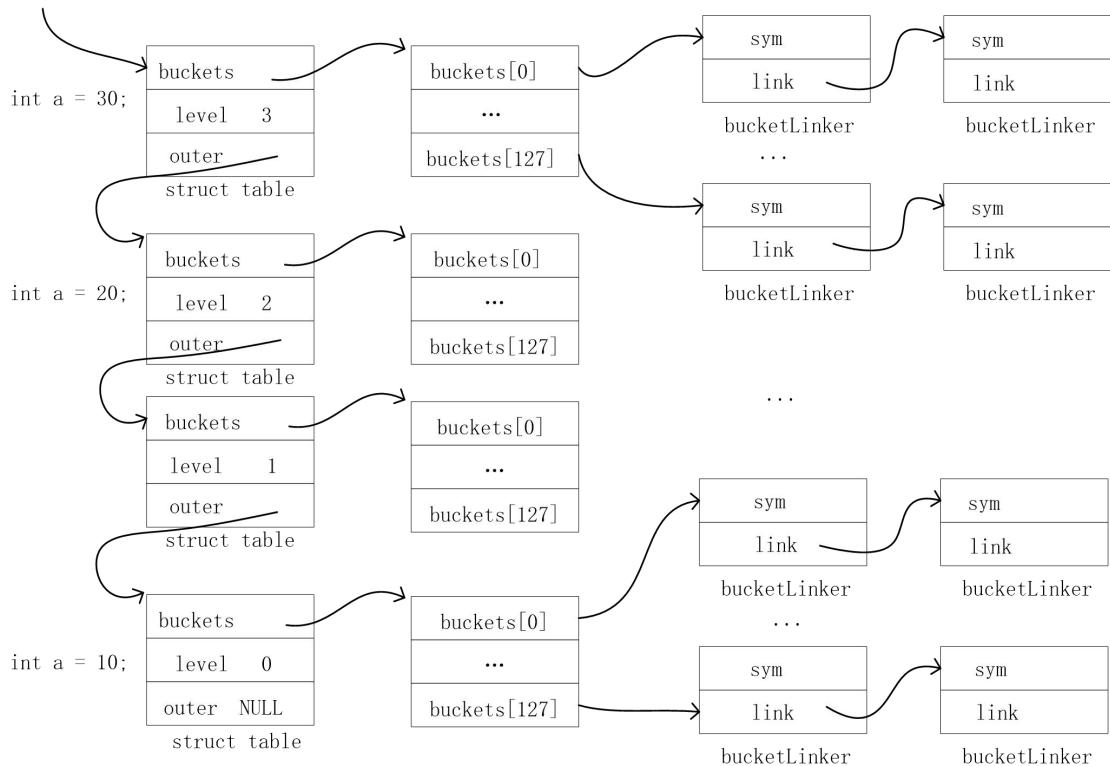


图 2.37 多个作用域的符号表

由图 2.37，我们可知，图 2.36 第 2 行的符号 `a` 是存放在深度为 0 的符号表中，而在深度为 1 的符号表中，我们并没有定义新的局部变量。图 2.36 第 5 行的符号 `a` 存放在深度为 2 的符号表中，而第 8 行的符号 `a` 则存放在深度为 3 的符号表中。当程序执行到图 2.36 的第 13 行时，当前符号表是 `level` 为 1 的符号表，第 13 行的语句 “`a = 60;`” 需要访问符号 `a`，我们要检查一下当前符号表中是否存在符号 `a`，如果没有，就通过 `outer` 指针查找外层的符号表，此时会在 `level` 为 0 的符号表中查找到全局变量 `a`。通过图 2.37 的多级符号表，我们实现了 C 语言中“作用域（scope）”的概念。

## 2.6 本章习题

1. UCC 编译器在进行词法分析时，用以下宏 `IS_EOF()` 来判断当前文件是否结束，请问是否可删去其中的条件 “`((cur)-Input.base) == Input.size`”。

```
//即是否可改为#define IS_EOF(cur) (*cur) == END_OF_FILE)
unsigned char END_OF_FILE = 255;
#define IS_EOF(cur) (*cur) == END_OF_FILE && ((cur)-Input.base) == Input.size)
```

2. 请仿照图 2.18 和 2.22，画出以下结构体 `struct Data` 和函数 `func()` 对应的类型结构。

```
struct Data{
    int a;
    int * ptr;
    float num[4];
};

double * func(int,float);
```

3. 分别用 gcc、g++、clang 和 ucc 命令来编译并运行以下程序，对比不同的实验结果。

```
int main(int argc,char * argv[]){
    typedef int INT32;
    typedef int INT32;
    struct INT32{
        int arr[4];
    };
    return 0;
}
```

# 第3章 语法分析

## 3.1 C 语言的表达式

### 3.1.1 条件表达式和二元表达式

从第3章开始，我们进入UCC编译器的语法分析阶段。相关的代码主要在ccl\decl.c、ccl\expr.c和ccl\stmt.c，分别对应声明、表达式和语句。在第1章时，我们已经通过一个简单的例子ucc\examples\sc对这些概念有了一个直观的体会，在本章中，我们需要结合C标准文法来进行语法分析。基本的分析方法和第1章的例子类似，只是所涉及到的文法更加复杂，所构建的语法树也更加庞大。可对照C89标准文法来阅读UCC的源代码。在本章中，我们要完成的任务很明确，就是把C源代码转换成更加适合编译器处理的语法树。我们需要先熟悉一下语法树上最基本的结点，如图3.1所示。

```

1  ****
2      抽象语法树结点的类别，共3大类：
3          与声明有关的、与表达式有关的、与语句有关的。
4      NK:      Node Kind
5  ****
6 enum nodeKind
7 { //与声明有关
8     NK_TranslationUnit,      NK_Function,           NK_Declaration,
9     NK_TypeName,             NK_Specifiers,         NK_Token,
10    NK_TypedefName,          NK_EnumSpecifier,       NK_Enumerator,
11    NK_StructSpecifier,      NK_UnionSpecifier,      NK_StructDeclaration,
12    NK_StructDeclarator,    NK_PointerDeclarator,   NK_ArrayDeclarator,
13    NK_FunctionDeclarator,  NK_ParameterTypeList, NK_ParameterDeclaration,
14    NK_NameDeclarator,      NK_InitDeclarator,     NK_Initializer,
15 //与表达式有关
16    NK_Expression,
17 //与语句有关
18    NK_ExpressionStatement, NK_LabelStatement,   NK_CaseStatement,
19    NK_DefaultStatement,    NK_IfStatement,        NK_SwitchStatement,
20    NK_WhileStatement,      NK_DoStatement,       NK_ForStatement,
21    NK_GotoStatement,       NK_BreakStatement,   NK_ContinueStatement,
22    NK_ReturnStatement,     NK_CompoundStatement
23 };
24 ****
25 kind:  the kind of node
26 next:  pointer to next node
27 coord: the coordinate of node
28 ****
29 #define AST_NODE_COMMON \
30     int kind;           \
31     struct astNode *next; \
32     struct coord coord;
33

```

```

34 typedef struct astNode
35 {
36     AST_NODE_COMMON
37 } *AstNode;

```

图 3.1 struct astNode

图 3.1 第 34 至 37 行的结构体 struct astNode 是语法树中最基本的结点，第 30 行的 kind 用于记录结点的类别，其取值范围由第 6 至第 23 行的枚举常量来确定，第 8 至 14 行的 NK\_Function 等枚举常量主要用于与声明相关的语法树结点，而第 16 行的 NK\_Expression 就用于表达式对应的语法树结点，第 18 至 22 行的 NK\_IfStatement 等主要用于与语句相关的结点。图 3.1 第 31 行的 next 域用于构成链表，第 32 行的 coord 主要用于记录与语法树结点对应的 C 源代码的位置。结构体 struct astNode 相当于所有语法树结点的父类，如果要描述表达式对应的结点，可引入结构体 struct astExpression，如图 3.2 所示。

```

1  ****
2  ty: expression type
3  op: expression operator
4  isArray: array or not
5  isfunc: function or not
6  lvalue: lvalue or not
7  bitfld: bit-field or not
8  inreg: in register or not
9  kids: expression operands
10 val: expression value
11 ****
12 struct astExpression{
13     AST_NODE_COMMON
14     Type ty;
15     int op : 16;
16     int isArray : 1;
17     int isfunc : 1;
18     int lvalue : 1;
19     int bitfld : 1;
20     int inreg : 1;
21     int unused : 11;
22     struct astExpression *kids[2];
23     union value val;
24 };

```

图 3.2 struct astExpression

图 3.2 第 14 行的 ty 用于记录表达式对应的类型，我们在 2.4 节中初步介绍过 C 语言的类型系统。第 15 行的 op 是 operator 的缩写，用于记录表达式的运算符，而第 22 行的 kids 则用于记录表达式的运算数。第 16 行的 isArray 用于标记当前结点是否为数组。本来第 14 行的 ty 域就已记录了结点的类型信息，但是由于数组名在 C 语言中有时会被当作“指向数组第 0 个元素的指针”，此时在 ty 域中记录的类型已被调整为指针类型，所以需要 isArray 来标记这个结点原本是个数组。第 18 行的 lvalue 则是 left value 的缩写，代表对应的表达式是否为左值。而第 19 行的 bitfld 用来标记当前结点是否为结构体对象的位域成员，对位域成员的访问比较特殊，需要进行移位操作。有了这个基础后，我们就可以来看一下表达式的语法分析函数，如图 3.3 所示。

```

1  /**
2  *  expression:
3  *      assignment-expression
4  *      expression , assignment-expression

```

```
5  */
6 AstExpression ParseExpression(void) {
7     AstExpression expr, comaExpr;
8
9     expr = ParseAssignmentExpression();
10    while(CurrentToken == TK_COMMA) {
11        CREATE_AST_NODE(comaExpr, Expression);
12        comaExpr->op = OP_COMMA;
13        comaExpr->kids[0] = expr;
14        NEXT_TOKEN;
15        comaExpr->kids[1] = ParseAssignmentExpression();
16        expr = comaExpr;
17    }
18    return expr;
19 }
```

图 3.3 ParseExpression()

由图 3.3 第 1 至 5 行注释中的产生式可知，C 语言的表达式由若干个被逗号隔开的赋值表达式构成。我们以下面“`a,b,c`”为例来说明一下由函数 `ParseExpression()` 所构成的语法树，如图 3.4 所示。

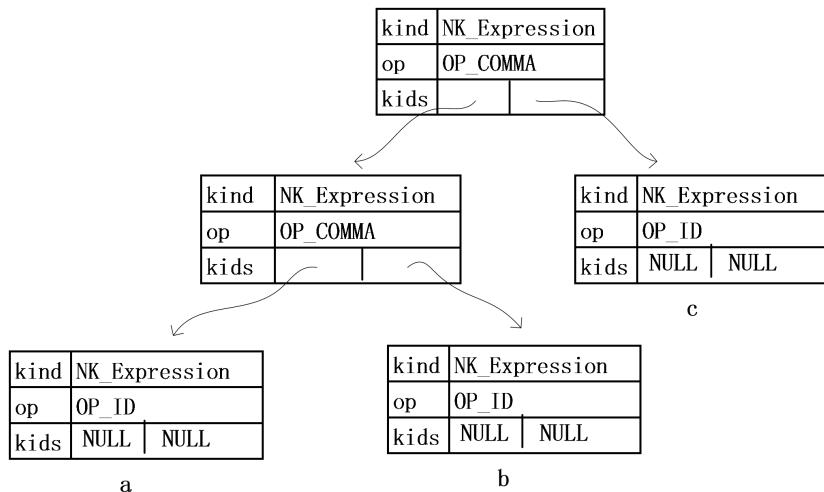


图 3.4 表达式 “ $a, b, c$ ” 对应的语法树

图 3.4 中 op 类别为 OP\_COMMA 的结点是通过图 3.3 第 11 至 15 行构建的，第 11 行的宏 CREATE\_AST\_NODE() 创建了一个 kind 域为 NK\_Expression 的结点，接下来的代码用于对该结点进行赋值。子表达式 a 对应的语法树结点会在图 3.3 第 9 行被构建，而子表达式 b 和子表达式 c 则通过第 15 行的 ParseAssignmentExpression() 函数来构建。图 3.5 给出了函数 ParseAssignmentExpression 的相关代码。

```
1  /**
2   * assignment-expression:
3   *     conditional-expression
4   *     unary-expression assignment-operator assignment-expression
5   * assignment-operator:
6   *     = *= /= %= += -= <<= >>= &= ^= |=
7   * There is a little twist here: the parser always treats the
8   * first nonterminal as a conditional expression.
9  */
10 AstExpression ParseAssignmentExpression(void) {
11   AstExpression expr;
12  /*****
```

```

13      It is not accurate here.
14      We will check it during semantics .
15      ****
16 expr = ParseConditionalExpression();
17 ****
18 see token.h
19      TOKEN(TK_ASSIGN,      "=")
20      TOKEN(TK_BITOR_ASSIGN, "|=")
21      .....
22      TOKEN(TK_MOD_ASSIGN,   "%=")
23 ****
24 if (CurrentToken >= TK_ASSIGN && CurrentToken <= TK_MOD_ASSIGN) {
25     AstExpression asgnExpr;
26     CREATE_AST_NODE(asgnExpr, Expression);
27     asgnExpr->op = BINARY_OP;
28     asgnExpr->kids[0] = expr;
29     NEXT_TOKEN;
30     asgnExpr->kids[1] = ParseAssignmentExpression();
31     return asgnExpr;
32 }
33 return expr;
34 }

```

图 3.5 ParseAssignmentExpression()

图 3.5 第 2 至 4 行的注释告诉我们，赋值表达式由“条件表达式”或者“一元表达式 赋值运算符 赋值表达式”构成，第 6 行的注释列出了“ $=$ ”等赋值运算符。而一元表达式只是条件表达式的一个特例，这就给语法分析带来了难题。在图 3.5 第 16 行处，我们实际上有两个候选式可用，我们没有办法根据当前读头下的 token 来决定是按条件表达式还是按一元表达式进行分析。这需要向前扫描多个 token 之后，例如遇到了表达式“ $a+3?b:c$ ”中的问号时，我们才能确定这应是个条件表达式。此处，UCC 采取的策略是通过修改文法来简化语法分析工作，图 3.5 中的代码实际上是按以下文法来进行语法分析的。

```

assignment-expression:
    conditional-expression
    conditional-expression assignment-operator assignment-expression

```

当然，修改后的文法所对应的语言就发生了变化，由此构建的语法分析器是不够准确的。例如，按照 C 标准文法，以下代码在语法上就是非法的。但 UCC 编译器修改了赋值表达式的文法后，会导致“(a?b:c)=d”被当作符合文法的赋值表达式。因此，在语法分析阶段，UCC 编译器并没有对以下 C 程序进行报错。

```

// hello.c
int a,b,c,d;
int main(int argc,char * argv[]){
    (a?b:c) = d;
    return 0;
}

```

不过，UCC 编译器会在语义检查时发现这个错误。如下所示的错误提示告诉我们，UCC 编译器是在 UCC 源代码 exprchk.c 的第 1196 行检测到这个错误的。文件 exprchk.c 中的代码是用来对表达式进行语义检查的。

```

iron@ubuntu:demo$ ucc hello.c -o hello
(exprchk.c,1196):(hello.c,4):error:The left operand cannot be modified

```

通过图 3.5 中的函数 ParseAssignmentExpression()，我们可以为赋值表达式“ $a=b=c$ ”构建一棵如图 3.6 所示的语法树。表达式“ $a=b=c$ ”的子表达式 a 对应的语法树结点由图 3.5

第 16 行调用的函数 ParseConditionalExpression() 构造，而子表达式 “ $b=c$ ” 对应的语法子树则由在第 30 行递归调用的函数 ParseAssignmentExpression 所构建。而图中 op 类型为 OP\_ASSIGN 的结点则由第 26 行的 CREATE\_AST\_NODE() 所创建，接下来的第 27 至 30 行则完成该结点的初始化。

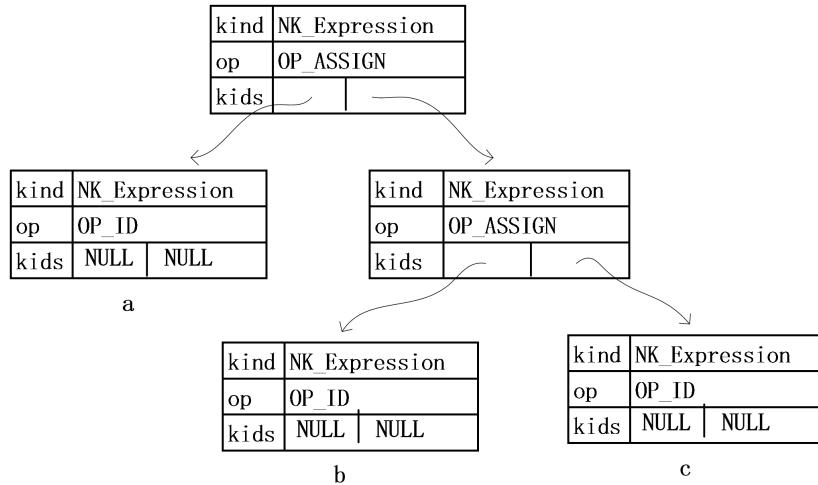


图 3.6 “ $a=b=c$ ” 对应的语法树

接下来，我们就来看一下 ParseConditionalExpression() 函数的代码，该函数完成了条件表达式的分析，如图 3.7 所示。

```

1 /**
2  * conditional-expression:
3  *     logical-OR-expression
4  *     logical-OR-expression ? expression : conditional-expression
5 */
6 static AstExpression ParseConditionalExpression(void) {
7     AstExpression expr;
8
9     expr = ParseBinaryExpression(Prec[OP_OR]);
10    if (CurrentToken == TK_QUESTION) {
11        AstExpression condExpr;
12
13        CREATE_AST_NODE(condExpr, Expression);
14        condExpr->op = OP_QUESTION;
15        condExpr->kids[0] = expr;
16        NEXT_TOKEN;
17
18        CREATE_AST_NODE(condExpr->kids[1], Expression);
19        condExpr->kids[1]->op = OP_COLON;
20        condExpr->kids[1]->kids[0] = ParseExpression();
21        Expect(TK_COLON);
22        condExpr->kids[1]->kids[1] = ParseConditionalExpression();
23
24        return condExpr;
25    }
26    return expr;
27 }
```

图 3.7 ParseConditionalExpression()

图 3.7 第 2 至 4 行给出了条件表达式的产生式，C 语言中有许多不同的二元运算符，在第 1.3 节时，我们就发现对不同的二元表达式而言，除了运算符有所区别外，其语法分析函

数的分析套路几乎都是一样的。UCC 编译器为了避免冗余的代码，统一采用 ParseBinaryExpression() 来处理不同的二元表达式，如图 3.7 第 9 行所示，函数调用时的参数 Prec[OP\_OR] 指明了“逻辑或”运算符的优先级。图 3.8 给出了上述代码为表达式“a?b:c”所构建的语法树。图 3.7 第 9 行调用的 ParseBinaryExpression() 函数创建了子表达式 a 对应的结点，而第 13 至 15 行则构造了图 3.8 中 op 域为 OP\_QUESTION 的结点，第 18 行至 22 行构建了 op 域为 OP\_COLON 的结点，子表达式 b 由第 20 行的 ParseExpression() 所创建，而子表达式 c 则由第 22 行的 ParseConditionalExpression() 所创建。

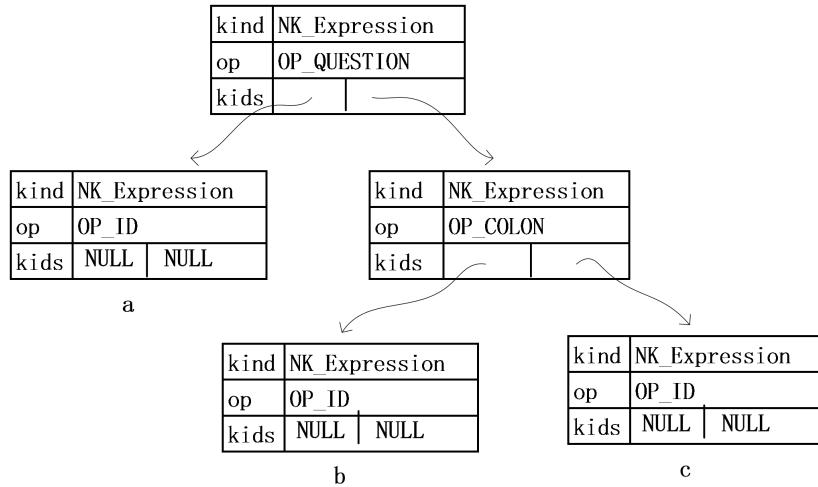


图 3.8 “a?b:c” 对应的语法树

紧接着，让我们来看看 C 标准文法中的二元表达式，如图 3.9 所示。图中越早出现的运算符，其运算符优先级越低，例如，“||”运算符是优先级最低的，其次是“&&”。图 3.9 第 1 行的“逻辑或表达式”由若干个“逻辑与表达式”进行“||”运算构成；而第 4 行的“逻辑与表达式”是由若干个“按位或表达式”进行“&&”运算构成的表达式。要得到“逻辑或表达式”，就要先得“逻辑与表达式”，这意味着运算符“&&”的优先级要比“||”更高。C 标准文法中为了表达不同二元运算符的优先级，定义了如图 3.9 所示的“套路相同但又有层次结构”的产生式。处于同一层次的运算符，其优先级是一样的，例如图 3.9 第 10 行的“+”和第 11 行的“-”。

```

1 logical-OR-expression:
2     logical-AND-expression
3     logical-OR-expression || logical-AND-expression
4 logical-AND-expression:
5     inclusive-OR-expression
6     logical-AND-expression && inclusive-OR-expression
7 .....
8 additive-expression:
9     multiplicative-expression
10    additive-expression + multiplicative-expression
11    additive-expression - multiplicative-expression
12 multiplicative-expression:
13    cast-expression
14    multiplicative-expression * cast-expression
15    multiplicative-expression / cast-expression
16    multiplicative-expression % cast-expression
  
```

图 3.9 二元运算符表达式

语法分析时，也可以为图 3.9 中的每个非终结符编写一个分析函数，但很快我们就会发现，这些分析函数几乎雷同。我们可以抽取这些不同函数的共性，把它们写到同一个函数

`ParseBinaryExpression()`中，它们的个性则通过函数的参数来进行区别。如果我们定义“逻辑或 $\|$ ”运算的优先级为4，则优先级更高的“逻辑与 $\&\&$ ”运算的优先级为5，值越大，表示优先级越高。若把图3.9第1行的logical-OR-expression用 $E_4$ 来表示，而第4行的logical-AND-expression用 $E_5$ 来表示，优先级为4的二元运算符记为 $BOP_4$ ，则图3.9第1至3行的产生式可简写以下形式：

$E_4:$   
 $E_5$   
 $E_4 \quad BOP_4 \quad E_5$

同理可以写出 $E_5$ 的产生式，一般地，我们可以写出优先级为n的二元运算符 $BOP_n$ 所对应的产生式 $E_n$ ，如下所示。这些产生式全部是采用左递归的形式，暗示着其中的二元运算符是左结合的。

$E_n:$   
 $E_{n+1}$   
 $E_n \quad BOP_n \quad E_{n+1}$

当然，C语言的二元运算符个数是有限的，由C的标准文法，当n为13时， $E_{n+1}$ （即 $E_{14}$ ）就不再是二元表达式了，而是强制类型转换表达式（cast-expression），如图3.9第12至16行所示。不过UCC把强制类型转换也当作一元运算符来处理，一起并入了产生式unary-expression中，如下所示：

```
unary-expression:  

postfix-expression //后缀表达式  

unary-operator unary-expression // ++ -- & * + - ! ~  

( type-name ) unary-expression //强制类型转换  

sizeof unary-expression //sizeof  

sizeof ( type-name ) //sizeof
```

这就使得 $E_{14}$ 表达式被当作一元表达式（unary-expression）来处理了，最终我们面对的不同优先级的二元表达式可表示为：

$E_4:$   
 $E_5$   
 $E_4 \quad BOP_4 \quad E_5$

$E_5:$   
 $E_6$   
 $E_5 \quad BOP_5 \quad E_6$   
 $\dots$

$E_{13}:$   
 $E_{14}$   
 $E_{13} \quad BOP_{13} \quad E_{14}$

结合这些产生式，我们可以写出形如图3.10的分析函数。图3.10第2行的函数参数prec表示当前正在分析的二元表达式的运算符优先级，对应上述递推公式 $E_n$ 的下标n。第3行的宏`HIGHEST_BIN_PREC`定义了优先级最高的二元运算符所对应的优先级，在二元运算符中，乘法的优先级最高，所以第2行的`Prec[OP_MUL]`为13。

```
1 //统一处理所有的二元运算符表达式
2 static AstExpression ParseBinaryExpression(int prec) {
3     #define HIGHEST_BIN_PREC    Prec[OP_MUL]
4     AstExpression binExpr;
5     AstExpression expr;
6
7     if(prec == HIGHEST_BIN_PREC) {
8         expr = ParseUnaryExpression();
9     }else{
10        expr = ParseBinaryExpression(prec+1);
11    }
```

```

12  while(IsBinaryOP(CurrentToken) && (Prec[BINARY_OP] == prec)) {
13      CREATE_AST_NODE(binExpr, Expression);
14      binExpr->op = BINARY_OP;
15      binExpr->kids[0] = expr;
16      NEXT_TOKEN;
17      if(prec == HIGHEST_BIN_PREC) {
18          binExpr->kids[1] = ParseUnaryExpression();
19      }else{
20          binExpr->kids[1] = ParseBinaryExpression(prec+1);
21      }
22      /***** Try to loop again to find whether binExpr could be *****
23      the left operand of the Current Operator when we are here.
24      *****/
25
26      expr = binExpr;
27  }
28  return expr;
29 }
```

图 3.10 ParseBinaryExpression()版本 1

当调用 ParseBinaryExpression(n)时, 表示我们要分析  $E_n$  对应的表达式。而表达式  $E_n$  由若干个  $E_{n+1}$  表达式进行  $BOP_n$  运算构成, 可用如下集合来表示。

$$E_n = \{E_{n+1}, E_{n+1} BOP_n E_{n+1}, E_{n+1} BOP_n E_{n+1} BOP_n E_{n+1}, \dots\}$$

图 3.10 第 7 至 11 行完成了  $E_{n+1}$  表达式的分析, 第 12 行判断一下当前的 token 是否为二元运算符  $BOP_n$ , 第 13 行创建了一个运算符结点, 而第 17 至 21 行实现了  $E_{n+1}$  表达式的分析, 接着我们再通过第 12 行的 while 循环判断一下是否还有二元运算符  $BOP_n$ 。

这个版本的 ParseBinaryExpression()已经把左结合的二元运算符进行统一处理了, 但其效率还可进一步改进。实际上, 由  $E_{13}$  生成的表达式总是以 unary-expression 开始的, 而  $E_{12}$  又是由若干个  $E_{13}$  进行  $BOP_{12}$  运算构成, 如果把  $E_{13}$  代入  $E_{12}$ , 则  $E_{12}$  也总是以 unary-expression 开始。以此类推, 可以发现由  $E_n$  生成的表达式总是以 unary-expression 开始的。按照这样的思路, UCC 中给出了一个更高效的 ParseBinaryExpression()函数, 如图 3.11 所示。

```

1 //统一处理所有的二元表达式
2 static AstExpression ParseBinaryExpression(int prec)
3 {
4     AstExpression binExpr;
5     AstExpression expr;
6     int newPrec;
7
8     expr = ParseUnaryExpression();
9     /// while the following binary operator's precedence is higher than current
10    /// binary operator's precedence, parses a higer precedence expression
11    while (IsBinaryOP(CurrentToken) && (newPrec = Prec[BINARY_OP]) >= prec)
12    {
13        CREATE_AST_NODE(binExpr, Expression);
14
15        binExpr->op = BINARY_OP;
16        binExpr->kids[0] = expr;
17        NEXT_TOKEN;
18
19        binExpr->kids[1] = ParseBinaryExpression(newPrec + 1);
20        expr = binExpr;
21    }
```

```

22     return expr;
23 }

```

图 3.11 ParseBinaryExpression() 版本 2

如果我们要调用 ParseBinaryExpression(4) 来对  $E_4$  进行分析，但是此时实际上遇到的表达式可能是 “unary-expression<sub>1</sub> BOP<sub>7</sub> unary-expression<sub>2</sub> BOP<sub>4</sub> unary-expression<sub>3</sub>”。由  $E_n$  生成的表达式总是以 unary-expression 开始，在图 3.11 第 8 行我们直接调用 ParseUnaryExpression() 来识别 unary-expression<sub>1</sub>，接下来我们遇到的是 BOP<sub>7</sub>，根据  $E_n$  的递推公式，在 BOP<sub>n</sub> 之后的应是  $E_{n+1}$ ，我们期望接下来的表达式是  $E_8$ ，因此，在图 3.11 第 11 行我们用 newPrec 记下优先级 7，而此时局部变量 prec 为 4，第 19 行的代码则调用 ParseBinaryExpression(newPrec+1) 来识别  $E_8$ 。在递归调用的 ParseBinaryExpression(8) 中，我们又会在第 8 行识别 unary-expression<sub>2</sub>，此时局部变量 prec 为 8，而当前遇到的 token 为 BOP<sub>4</sub>，即第 11 行的条件不成立，我们从 ParseBinaryExpression(8) 函数中返回，会回到 ParseBinaryExpression(4) 中，对 ParseBinaryExpression(4) 这个函数而言，其局部变量 prec 仍然为 4，面对读头下的 BOP<sub>4</sub>，此时 newPrec 为 4，则第 11 行的条件再次成立，我们期望紧跟在 BOP<sub>4</sub> 后面的表达式是  $E_5$ 。在第 19 行调用的 ParseBinaryExpression(5) 函数则完成了 unary-expression<sub>3</sub> 的识别。图 3.12 给出了两种不同版本的 ParseBinaryExpression 函数所形成的分析树。

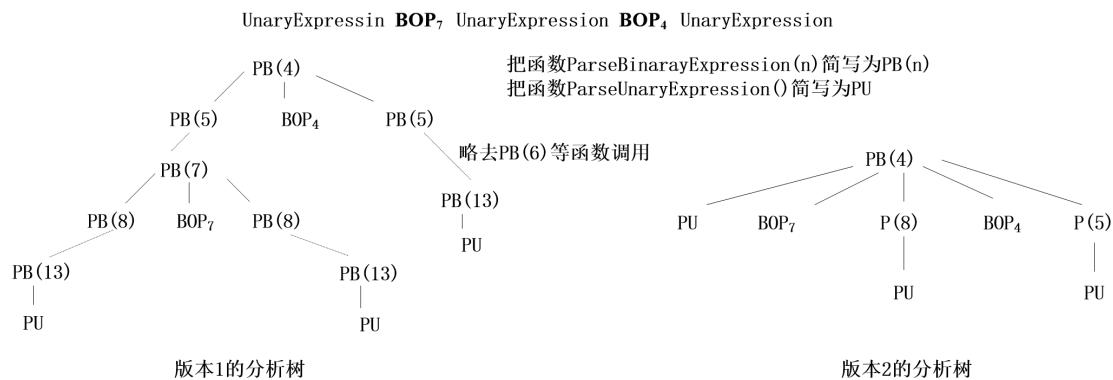


图 3.12 两种不同版本的 ParseBinaryExpression 对应的分析树

我们会在下一小节对 ParseUnaryExpression() 函数进行讨论。

### 3.1.2 一元表达式、后缀表达式和基本表达式

在这一小节，我们来分析一下一元运算表达式，对应的分析函数 ParseUnaryExpression() 在 ucl\expr.c 中，如图 3.13 所示。由图 3.13 第 2 至 7 行的产生式可知，非终结符 unary-expression 有多个候选式，第 16 至 29 行用来分析候选式 “unary-operator unary-expression” 。对于表达式 “\*ptr++” 而言，由第 3 行和第 4 行的产生式可知，只有分析完了后缀表达式 “ptr++” 之后，我们才能得到一元运算符 “\*” 的操作数，进而构成完整的一元表达式 “\*ptr++” 。这就意味着在 C 语言中，形如 “++” 这样的后缀运算符的优先级要比一元运算符更高。当然，运算符 “++” 有 “前加加” 和 “后加加” 的区别，例如 “++ptr” 和 “ptr++”，其中，“前加加”是一元运算符，而“后加加”是后缀运算符。在 C 语言的运算符优先级表格中，一般把 “++” 统一划入 “一元运算符” 中，再规定一元运算符要按从右到左的顺序进行运算，这其实也相当于 “后加加” 要比其他一元运算符更优先计算。而从产生式的角度来理解优先级则很简单，即后缀运算符要先于一元运算符进行计算。同样道理，由 C 语言标准文法，二元表达式是由一元表达式构成，这意味着只有完成了一元表达式的计算，才能进行二元表达式的计算，即所有的一元运算符的优先级比二元运算符更高。简单而言，在 C 语言运算

符的优先级上，可粗粗地记为：

后缀运算符 > 一元运算符 > 二元运算符

关于运算符的优先级，曾经遇到如下所示的一个 Bug，某嵌入式 C 程序员本想通过 `(*ptr)++` 来统计脉冲的个数，不过却写成了 `*ptr++`，这两者有完全不同的语义。这样的 Bug 埋没在几万行的 C 代码中，看着嵌入式产品每次上电一段时间后就跑飞，实在是让人郁闷的一件事情。

```
while(1){
    *ptr = 0;
    .....
    while(flag){
        *ptr++; //本来应是(*ptr)++;
        .....
    }
}
```

至于不同二元运算符的优先级，估计大部分写了 N 年代码的 C 程序员都记不全，我们就按《Thinking in C++》作者 Bruce Eckel 的建议来做，就是多用小括号。

```
1  ****
2  * unary-expression:
3  *     postfix-expression
4  *     unary-operator unary-expression
5  *     ( type-name ) unary-expression
6  *     sizeof unary-expression
7  *     sizeof ( type-name )
8 *
9  * unary-operator:
10 *     ++ -- & * + - ! ~
11 ****
12 static AstExpression ParseUnaryExpression(){
13     AstExpression expr;
14     int t;
15     switch (CurrentToken){
16         // unary-expression : unary-operator unary-expression
17     case TK_INC:           // ++num
18     case TK_DEC:           // --num
19     case TK_BITAND:        // &num
20     case TK_MUL:           // *ptr
21     case TK_ADD:           // +num
22     case TK_SUB:           // -num
23     case TK_NOT:           // !flag
24     case TK_COMP:          // ~num
25         CREATE_AST_NODE(expr, Expression);
26         expr->op = UNARY_OP;
27         NEXT_TOKEN;
28         expr->kids[0] = ParseUnaryExpression();
29         return expr;
30     // 略
```

图 3.13 ParseUnaryExpression()

在图 3.13 中，第 2 至 6 行右递归的产生式，实际上暗示了一元运算符的右结合，即要按从右到左的顺序进行计算。由图 3.13 可知，如果当前 token 是第 17 至 24 行所列的一元运算符时，我们会执行第 25 至 29 行的代码。例如，对于表达式 “`&a`” 来说，通过调用 `ParseUnaryExpression` 函数，我们会为之创建一棵如图 3.14 所示的语法树。

The diagram illustrates a stack frame 'a' in two states. The top state shows a frame with three fields: 'kind' (NK\_Expression), 'op' (OP\_ADDRESS), and 'kids' (pointing to the bottom frame). The bottom state shows a frame with three fields: 'kind' (NK\_Expression), 'op' (OP\_ID), and 'kids' (NULL, NULL). An arrow points from the 'kids' field of the top frame to the 'a' label.

kind	NK_Expression
op	OP_ADDRESS
kids	

kind	NK_Expression	
op	OP_ID	
kids	NULL	NULL

图 3.14 &amp;a 对应的语法树

图 3.13 第 28 行的 ParseUnaryExpression() 是个递归调用，所以我们需要相应的递归出口，由图 3.13 第 2 至 7 行的产生式，我们可以发现，非终结符 unary-expression 的候选式 sizeof(typename) 和 postfix-expression 就是递归定义的出口。与这两个候选式对应的代码如图 3.15 所示。

```

1 //static AstExpression ParseUnaryExpression(){
2 //.....
3 case TK_SIZEOF:
4     CREATE_AST_NODE(expr, Expression);
5     expr->op = OP_SIZEOF;
6     NEXT_TOKEN;
7     if (CurrentToken == TK_LPAREN) {
8         BeginPeekToken();
9         t = GetNextToken();
10        if (IsTypeName(t)) { // sizeof (type-name)
11            EndPeekToken();
12            NEXT_TOKEN;
13            expr->kids[0] = (AstExpression)ParseTypeName();
14            Expect(TK_RPAREN);
15        } else{ // sizeof unary-expression
16            EndPeekToken();
17            expr->kids[0] = ParseUnaryExpression();
18        }
19    }else{
20        // sizeof unary-expression
21        expr->kids[0] = ParseUnaryExpression();
22    }
23    return expr;
24 default:
25     return ParsePostfixExpression();
26 }
```

图 3.15 ParseUnaryExpression()\_sizeof

图 3.15 第 25 行调用 ParsePostfixExpression() 对后缀表达式进行分析；第 3 至 23 行则对由一元运算符 sizeof 构成的表达式进行分析。按 C 的语法，当类型名充当 sizeof 的操作数时，需要加上一对小括号，因此如下所列的 sizeof 表达式中 “sizeof int” 是非法的。而当一元表达式作为 sizeof 的操作数时，则不一定要加上小括号，如下所示的 sizeof a 和 sizeof(a) 都是合法的。

```

int a;
typedef int bbb;
sizeof(a);
sizeof a;
```

```
sizeof(int);
sizeof(bbb);
//sizeof int;
```

在图 3.15 第 8 行，我们遇到的问题是：在识别完“`sizeof(bbb)`”的前缀“`sizeof()`”之后，我们要判断一下接下来的标志符 `bbb` 是一个类型名，还是一个普通的变量名 `bbb`。图 3.15 第 10 行的 `IsTypeName()` 完成了这个判断。在 C 语言中，我们可以通过“`typedef int bbb;`”来为整型 `int` 取一个别名 `bbb`，在语法分析阶段，完整的类型系统还没有建立起来，此时我们只需要记录标志符 `bbb` 为一个类型名就可以了，UCC 编译器会将这些信息记录到向量 `TypedefNames` 中。

```
static Vector TypedefNames, OverloadNames;
```

但是对下面的 C 程序则言，在以下函数 `g` 内，标志符 `ccc` 又可当作形参来用，而不是用作一个类型名，只有离开函数 `g` 之后，标志符 `ccc` 才会再次充当类型名。因此，我们需要记录标志符 `ccc` 在函数 `g` 内已被另作他用，UCC 编译器会把这个信息记录到向量 `OverloadNames` 中。

```
typedef int ccc;      //ccc 是类型 int 的别名，作用域深度为 0
void g(int ccc){      //在函数 g 内，只是一个普通的变量名，作用域深度为 1
}
ccc c;                //ccc 是类型 int 的别名
```

UCC 编译器使用以下结构体 `struct tdname` 来描述由 `typedef` 所创建类型名的相关信息，如下所示，其中 `id` 域用于指向形如“`ccc`”的由 `typedef` 得来的类型名，`level` 记录了 `typedef` 语句所处作用域的深度，对上述“`typedef int ccc;`”而言，其深度为 0。当进入函数 `g`，遇到形参“`int ccc`”时，我们把 `overload` 域置为 1，表示在函数 `g` 内部 `ccc` 已经不再充当类型名，而用 `overloadLevel` 来记录当前所处的作用域深度为 1。

```
typedef struct tdname{
    char *id;          //类型名，例如 ccc
    int level;         //深度
    int overload;      //在上述函数 g 内，ccc 就不再是个类型名
    int overloadLevel; //发生“重载”时，所处的深度
} *TDName;
```

我们再举一个例子来说明结构体 `tdname` 中 `level` 域的作用，如下所示。UCC 编译器中有一个全局变量 `Level`，用来记录当前所处作用域的深度，全局作用域的深度为 0。在作用域深度为 1 的函数 `f` 内部，我们通过 `typedef` 定义了一个新的类型，此时 UCC 编译器会为之创建一个 `tdname` 对象，其 `level` 域为 1，并将该对象保存在向量 `TypedefNames` 中。当我们离开函数 `f`，再次进入全局作用域时，类型 `Data` 对应的 `tdname` 对象仍然保留在向量 `TypedefNames` 中，但其 `level` 值 1 大于当前作用域的深度值 0，即在函数 `f` 中定义的类型名 `Data` 在全局作用域是不可见的。

```
// Level 为 0
void f(void){
    //Level 为 1
    typedef struct{
        int a[4];
    } Data;
}
// Level 为 0
Data dt;      //报错，Data 不被视为一个类型名
int main(int argc,char * argv[]){
    return 0;
}
```

函数 `IsTypename()` 的代码如图 3.16 所示。C 语言自带的 `int` 和 `double` 等关键字是类型名，

而通过 `typedef` 也可以得到自定义的类型别名。在图 3.16 第 3 行，当面对一个标志符时，我们调用 `IsTypedefName()` 函数，判断该标志符是不是通过 `typedef` 关键字定义的类型名。图 3.16 第 8 至 13 行的代码会遍历向量 `TypedefNames` 中的每个 `tdname` 对象，当第 11 行的条件成立时，就认为该标志符确实是一个由 `typedef` 定义的类型别名。第 11 行的 “`tn->id == id`” 比较两个标志符的名字是否相同。在 “第 2.5 节 UCC 符号表管理” 时我们介绍过，对于同名的标志符，UCC 只在内存中记录一份标志符的名字，此处我们只要比较两个字符串指针是否相等即可。第 11 行的 “`tn->level <= Level`” 用来判断类型名在当前作用域中是否可见，例如上述 “`Data dt;`” 中的 `Data` 就不满足这个条件。而第 11 行的 “`! tn->overload`” 则用来判断类型名是否被重载，例如在上述函数 `g()` 中，标志符 `ccc` 就被当作变量名之用。

```

1 // 是否为在类型名: 包括 int 等基本类型和由 typedef 定义的类型名
2 int IsTypeName(int tok){
3     return tok == TK_ID ? IsTypedefName(TokenValue.p)
4             : (tok >= TK_AUTO && tok <= TK_VOID);
5 }
6 // 是否为通过 typedef 定义的类型名
7 static int IsTypedefName(char *id) {
8     Vector v = TypedefNames;
9     TDName tn;
10    FOR_EACH_ITEM(TDName, tn, v)
11        if (tn->id == id && tn->level <= Level && ! tn->overload)
12            return 1;
13    ENDFOR
14    return 0;
15 }
```

图 3.16 IsTypename()

对于一元表达式 `sizeof(a)`，通过图 3.15 的分析函数，我们可为之创建一棵如图 3.17 所示的语法树。

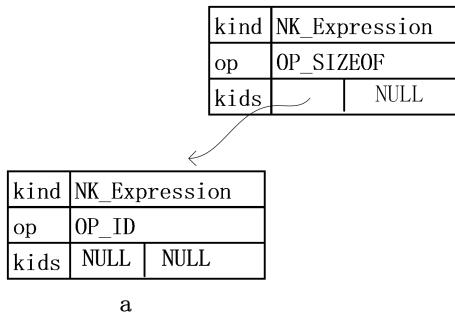


图 3.17 sizeof(a) 对应的语法树

一元表达式还有一个候选式 “( type-name ) unary-expression”，该候选式描述了 C 语言中的强制类型转换，相应的分析代码如图 3.18 所示。

```

1 //static AstExpression ParseUnaryExpression() {
2     case TK_LPAREN:
3         /// When current token is (, it may be a type cast expression
4         /// or a primary expression, we need to look ahead one token,
5         /// if next token is type name, the expression is treated as
6         /// a type cast expression; otherwise a primary expression
7         BeginPeekToken();
8         t = GetNextToken();
9         if (IsTypeName(t)) {
10             EndPeekToken();
11             CREATE_AST_NODE(expr, Expression);
```

```

12         expr->op = OP_CAST;
13         NEXT_TOKEN;
14         expr->kids[0] = (AstExpression) ParseTypeName();
15         Expect(TK_RPAREN);
16         expr->kids[1] = ParseUnaryExpression();
17         return expr;
18     } else{
19         EndPeekToken();
20         return ParsePostfixExpression();
21     }
22     break;

```

图 3.18 ParseUnaryExpression()\_cast

在 C 语言中, 形如“(int)a”的强制类型转换是以左括号开头的, 但是对非终结符 unary-expression 而言, 它的候选式“postfix-expression”和“( type-name ) unary-expression”都可能以左括号开始。在图 3.18 第 7 行处, 我们要判断应选用哪个候选式, 这需要再看一下紧跟在左括号的 token 是不是类型名, 如果是类型名, 我们就要按候选式“(type-name) unary-expression”去分析; 如果不是, 则词法分析器需要回溯到左括号处, 这就是图 3.18 第 7 行 BeginPeekToken() 和第 10 行 EndPeekToken() 的作用。通过图 3.18 第 11 至 16 行的代码, 我们可为表达式“(int)a”创建一个如图 3.19 所示的语法树, 第 14 行的 ParseTypeName() 用于分析类型名, 我们会在后续章节讨论 C 语言的声明时对其进行讨论。

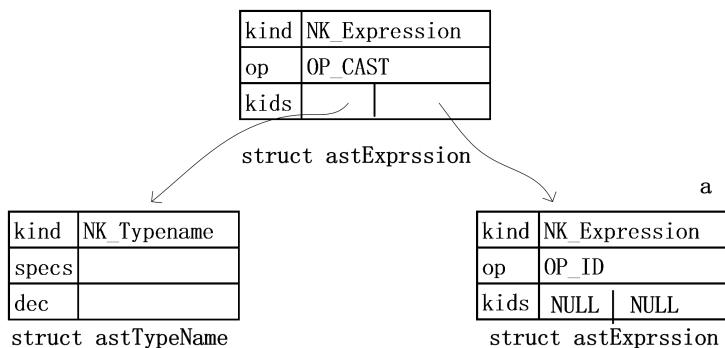


图 3.19 (int) a 对应的语法树

分析完 ParseUnaryExpression(), 让我们再来看一下后缀表达式对应的分析函数 ParsePostfixExpression() 的部分代码。如图 3.20 所示。

```

1 /**
2 * postfix-expression:
3 *     primary-expression
4 *     postfix-expression [ expression ]
5 *     postfix-expression ( [argument-expression-list] )
6 *     postfix-expression . identifier
7 *     postfix-expression -> identifier
8 *     postfix-expression ++
9 *     postfix-expression --
10 */
11 static AstExpression ParsePostfixExpression(void){
12     AstExpression expr, p;
13
14     expr = ParsePrimaryExpression();
15     while (1){
16         switch (CurrentToken){
17             case TK_LBRACKET: // postfix-expression [ expression ]

```

```

18     CREATE_AST_NODE(p, Expression);
19     p->op = OP_INDEX;
20     p->kids[0] = expr;
21     NEXT_TOKEN;
22     p->kids[1] = ParseExpression();
23     Expect(TK_RBRACKET);
24     expr = p;
25     break;
26 //略
27 }
28 }
29 }

```

图 3.20 ParsePostfixExpression()

由图 3.20 第 2 至 9 行，我们可以发现，所有的后缀表达式都是以基本表达式开头的，后面会跟上若干个后缀运算符。C 语言中定义了这么几种不同的后缀运算符：“数组下标[]”、“函数调用()”、“成员选择.”、“成员选择->”、“后加加++”和“后减减--”。图 3.20 第 17 至 25 行用于分析“后缀运算符[]”，而对其他后缀运算符进行分析时所用的套路与“[]”类似。这里就不再重复，仅给出与下面几个后缀表达式对应的语法树，如图 3.21 所示。

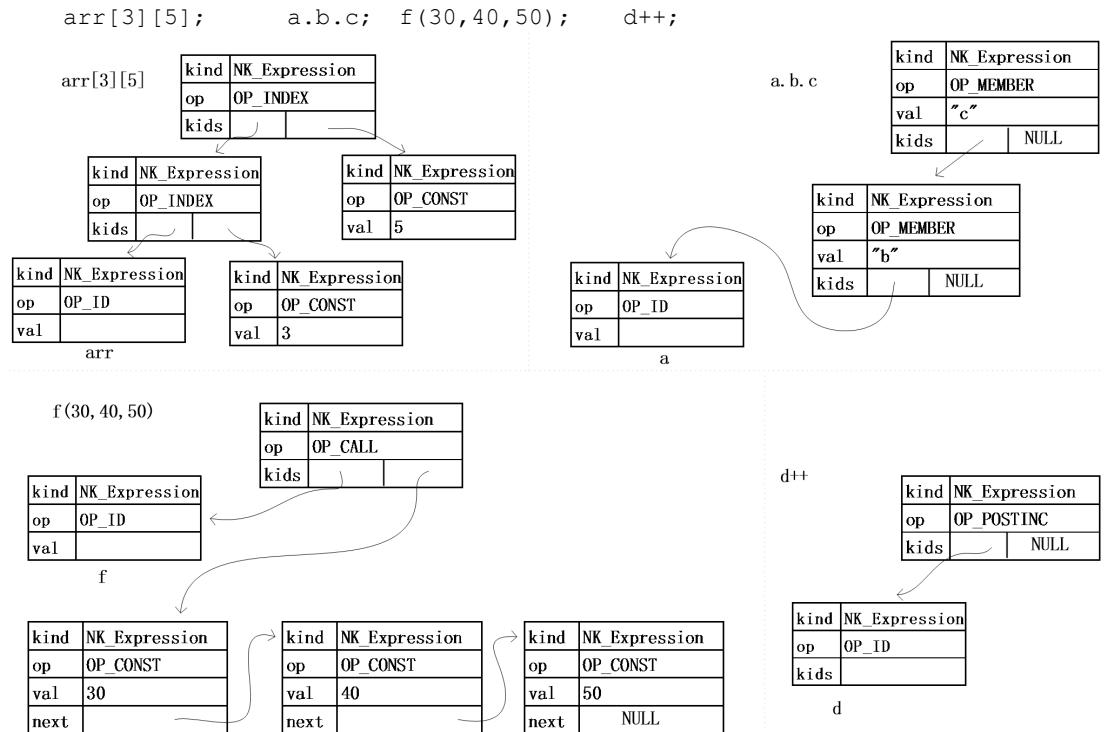


图 3.21 后缀运算符对应的语法树

最后，让我们来讨论一下基本表达式 PrimaryExpression，图 3.22 给出了与之相关的分析函数 ParsePrimaryExpression()。由图 3.22 第 2 至 6 行可知，基本表达式由标志符 ID、常量、字符串和加括号的表达式构成。图 3.22 第 12 至 17 行的代码为标志符构造了一个语法树结点，其 op 域记为 OP\_ID。图 3.22 第 18 至 35 行为整数和浮点数构建了语法树结点，第 32 行置其 op 域为 OP\_CONST，第 33 行把常量的值记录到 val 域，而整型和浮点数常量的类型则记录到 ty 域。图 3.22 第 36 至 44 行为字符串或宽字符串创建了语法树结点，其 op 域为 OP\_STR，其类型为字符数组，而第 45 至 49 行则用于分析候选式 (expression)。

```

1 /**
2 * primary-expression:
3 *      ID

```

```

4   *      constant
5   *      string-literal
6   *      ( expression )
7 */
8 static AstExpression ParsePrimaryExpression(void) {
9   AstExpression expr;
10
11  switch (CurrentToken) {
12  case TK_ID:
13    CREATE_AST_NODE(expr, Expression);
14    expr->op = OP_ID;
15    expr->val = TokenValue;
16    NEXT_TOKEN;
17    return expr;
18  case TK_INTCST:           // 12345678
19  case TK_UINTCST:         // 12345678U
20  case TK_LONGCST:         // 12345678L
21  case TK ULONGCST:        // 12345678UL
22  case TK_LLONGCST:        // 12345678LL
23  case TK_ULLONGCST:       // 12345678ULL
24  case TK_FLOATCST:        // 123.456f 123.456F
25  case TK_DOUBLECST:       // 123.456
26  case TK_LDOUBLECST:      // 123.456L 123.456L
27    CREATE_AST_NODE(expr, Expression);
28    if (CurrentToken >= TK_FLOATCST) {
29      CurrentToken++;
30    }
31    expr->ty = T(INT + CurrentToken - TK_INTCST);
32    expr->op = OP_CONST;
33    expr->val = TokenValue;
34    NEXT_TOKEN;
35    return expr;
36  case TK_STRING:          // "ABC"
37  case TK_WIDESTRING:      // L"ABC"
38    CREATE_AST_NODE(expr, Expression);
39    expr->ty = ArrayOf(((String)TokenValue.p)->len + 1,
40                         CurrentToken == TK_STRING ? T(CHAR) : WCharType);
41    expr->op = OP_STR;
42    expr->val = TokenValue;
43    NEXT_TOKEN;
44    return expr;
45  case TK_LPAREN:          // (expression)
46    NEXT_TOKEN;
47    expr = ParseExpression();
48    Expect(TK_RPAREN);
49    return expr;
50  default:
51    Error(&TokenCoord, "Expect identifier, string, constant or ()");
52    return Constant0;
53  }
54 }

```

图 3.22 ParsePrimaryExpression()

需要注意的是，在目前构造的表达式语法树上，只有 OP\_CONST 和 OP\_STR 对应结点

的类型信息是完整的,语法树中其他结点的类型信息需要在语义分析时通过符号表并结合类型规则进行推导。图 3.23 给出了基本表达式(a+2015+b)对应的语法树,其中只有常数 2015 对应结点的类型已填为 int,而其他结点的类型信息还未设置,在语义检查时,我们通过查找符号表,可为标志符 a 和 b 的结点补上类型信息,而整个表达式(a+2015+b)的类型则要根据相应的算术运算规则去推导。

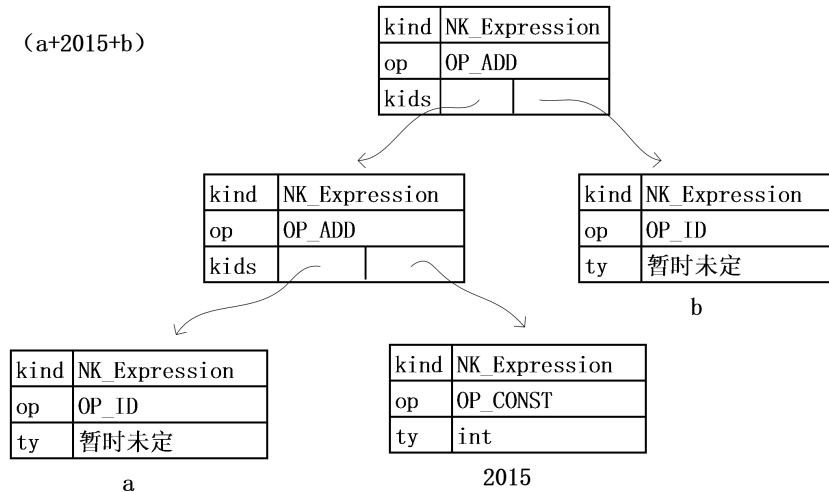


图 3.23 (a+2015+b) 对应的语法树

### 3.2 C 语言的语句

在这一节中,我们对 C 语言的各种语句进行分析,相关代码在 ucl\stmt.c 中。结构体 struct astNode 是语法树所有结点的父类,而 if 语句等需要在语法树结点上记录更多的信息,所以在 ucl\stmt.h 中定义了 struct astIfStatement 等结构体,我们会在讨论具体的语句时,在语法树示意图上画出相应的结构体。图 3.24 给出了语句的分析函数 ParseStatement()。

```

1  /**
2   * statement:
3   *     expression-statement
4   *     labeled-statement
5   *     case-statement
6   *     default-statement
7   *     if-statement
8   *     switch-statement
9   *     while-statement
10  *    do-statement
11  *    for-statement
12  *    goto-statement
13  *    continue-statement
14  *    break-statement
15  *    return-statement
16  *    compound-statement
17  */
18 static AstStatement ParseStatement(void) {
19   switch (CurrentToken) {
20   case TK_ID:
21     //*****
22     (1)
23     loopAgain : //...
  
```

```

24           goto loopAgain
25           (2)
26           f(20,30)
27           *****/
28       return ParseLabelStatement();
29   case TK_CASE:
30       return ParseCaseStatement();
31   case TK_DEFAULT:
32       return ParseDefaultStatement();
33   case TK_IF:
34       return ParseIfStatement();

```

图 3.24 ParseStatement()

图 3.24 第 2 至 16 行给出了语句的产生式，包括第 3 行的表达式语句

(expression-statement)、第 4 行的带标号的语句 (labeled-statement)、一直到第 16 行的复合语句 (compound-statement)。在 C 语言中，在表达式的后面加一个分号，就可以构成一个表达式语句。表达式语句由“expression ;”构成，其中不包含非终结符 statement，这意味着表达式语句可以充当递归定义 statement 的递归出口。类似的，break 语句、continue 语句、goto 语句和 return 语句的定义中都没有非终结符 statement，这些语句都可以作为递归定义 statement 的递归出口。一个没有出口的递归会陷入死循环，就如“从前有座山，山上有座庙，庙里有个老和尚，老和尚给小和尚讲故事说，从前有座山，山上有座庙，……”。

图 3.24 第 19 行根据当前读头下的 token，来决定要按哪个语句去做语法分析，第 29 至 34 行分别对 case 语句、default 语句和 if 语句进行分析。在当前记号是 TK\_ID 时，我们又会在第 20 行遇到老问题，以 TK\_ID 开始的句子可能是形如第 23 行的标号语句，也可能是形如第 26 行的表达式语句。我们需要再往前多读若干个 token 才能把它们区别开。在图 3.24 第 28 行，我们暂且调用 ParseLabelStatement()，在该函数内部我们会尝试再多读一个 token，若遇到冒号则说明我们面对的是标号语句；否则就是表达式语句，此时需要让词法分析器作回溯，对应的代码如图 3.25 第 21 至 34 行所示。

```

1 /**
2  * expression-statement:
3  *     [expression] ;
4 */
5 static AstStatement ParseExpressionStatement(void) {
6     AstExpressionStatement exprStmt;
7     CREATE_AST_NODE(exprStmt, ExpressionStatement);
8     if (CurrentToken != TK_SEMICOLON) {
9         exprStmt->expr = ParseExpression();
10    }
11    Expect(TK_SEMICOLON);
12    return (AstStatement)exprStmt;
13 }
14 /**
15 * label-statement:
16 *     ID : statement
17 */
18 static AstStatement ParseLabelStatement(void) {
19     AstLabelStatement labelStmt;
20     int t;
21     BeginPeekToken();
22     t = GetNextToken();
23     if (t == TK_COLON) {

```

```

24     EndPeekToken();
25     CREATE_AST_NODE(labelStmt, LabelStatement);
26     labelStmt->id = TokenValue.p;
27     NEXT_TOKEN;
28     NEXT_TOKEN;
29     labelStmt->stmt = ParseStatement();
30     return (AstStatement)labelStmt;
31 } else{
32     EndPeekToken();
33     return ParseExpressionStatement();
34 }
35 }

```

图 3.25 ParseLabelStatement()

图 3.25 第 25 行创建了一个类型为 struct astLabelStatement 的语法树结点，其 kind 域为 NK\_LabelStatement，而标号的名称在第 26 行用 id 域来记录，通过在第 29 行递归调用的 ParseStatement() 函数来对语句进行分析，所得的语法树记录到 stmt 域中。而图 3.25 第 33 行调用函数 ParseExpressionStatement() 来对表达式语句进行分析，对应的代码在第 5 至 13 行，第 7 行创建了一个 struct astExpressionStatement 的语法树结点，第 9 行调用 ParseExpression() 完成了表达式的语法分析，并把所构建的语法树记录到 expr 域中。图 3.26 给出了一个具体的例子及由上述代码构建的语法树。

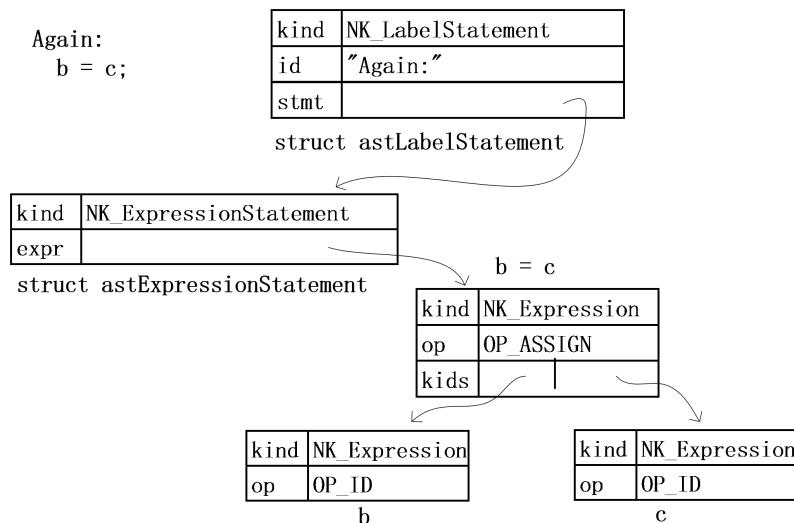


图 3.26 标号语句对应的语法树

而 case 语句和 default 语句的处理类似于标号语句，相关代码在 uclstmt.c 的函数 ParseCaseStatement() 和 ParseDefaultStatement() 中，这里不再啰嗦。接下来让我们看一下 if 语句和 switch 语句的分析函数，在完成表达式语法分析的前提下，我们可以发现，if 语句和 switch 语句的语法分析比我们预想的要简单得多，如图 3.27 所示。

```

1 /**
2 * if-statement:
3 *      if ( expression ) statement
4 *      if ( expression ) statement else statement
5 */
6 static AstStatement ParseIfStatement(void){
7     AstIfStatement ifStmt;
8
9     CREATE_AST_NODE(ifStmt, IfStatement);
10    NEXT_TOKEN;

```

```

11 Expect(TK_LPAREN);
12 ifStmt->expr = ParseExpression();
13 Expect(TK_RPAREN);
14 ifStmt->thenStmt = ParseStatement();
15 if (CurrentToken == TK_ELSE) {
16     NEXT_TOKEN;
17     ifStmt->elseStmt = ParseStatement();
18 }
19 return (AstStatement)ifStmt;
20 }
21 /**
22 * switch-statement:
23 *      switch ( expression ) statement
24 */
25 static AstStatement ParseSwitchStatement(void) {
26 AstSwitchStatement swtchStmt;
27
28 CREATE_AST_NODE(swtchStmt, SwitchStatement);
29 NEXT_TOKEN;
30 Expect(TK_LPAREN);
31 swtchStmt->expr = ParseExpression();
32 Expect(TK_RPAREN);
33 swtchStmt->stmt = ParseStatement();
34 return (AstStatement)swtchStmt;
35 }

```

图 3.27 if 语句和 switch 语句的语法分析

图 3.27 第 9 行创建了一个类型为 struct astIfStatement 的语法树结点，在第 12 行通过调用 ParseExpression() 来分析 if 语句中的表达式，把所得的语法子树记录到 expr 域中，对语句的分析仍然是通过在第 14 行和第 17 行调用 ParseStatement() 来进行，相应的语法子树则分别记录到 thenStmt 和 elseStmt 中。第 10 行的 NEXT\_TOKEN 只是用来跳过 TK\_IF，第 11 行的 Expect(TK\_LPAREN) 表示我们期望紧跟在 if 之后的是一个左括号。而图 3.27 第 21 至 35 行的代码则用于进行 switch 语句的分析。类似的，我们可以在 ucl\stmt.c 中，看到对其他语句的分析，所用的分析方法跟 ParseIfStatement() 并无多大区别，这里我们就仅以复合语句为例，如图 3.28 所示。

```

1 /**
2 * compound-statement:
3 *      { [declaration-list] [statement-list] }
4 * declaration-list:
5 *      declaration
6 *      declaration-list declaration
7 * statement-list:
8 *      statement
9 *      statement-list statement
10 */
11 AstStatement ParseCompoundStatement(void) {
12 AstCompoundStatement compStmt;
13 AstNode *tail;
14
15 Level++;
16 CREATE_AST_NODE(compStmt, CompoundStatement);
17 NEXT_TOKEN;

```

```

18 tail = &compStmt->decls;
19 while (CurrentTokenIn(FIRST_Declaration)) {
20     // for example, "f(20,30);", f is an id;
21     // but f(20,30) is a statement, not declaration.
22     if (CurrentToken == TK_ID && ! IsTypeName(CurrentToken))
23         break;
24     *tail = (AstNode) ParseDeclaration();
25     tail = &(*tail)->next;
26 }
27 tail = &compStmt->stmts;
28 while (CurrentToken != TK_RBRACE && CurrentToken != TK_END) {
29     *tail = (AstNode) ParseStatement();
30     tail = &(*tail)->next;
31     if (CurrentToken == TK_RBRACE)
32         break;
33     SkipTo(FIRST_Statement, "the beginning of a statement");
34 }
35 Expect(TK_RBRACE);
36 PostCheckTypedef();
37 Level--;
38 return (AstStatement) compStmt;
39 }

```

图 3.28 ParseCompoundStatement()

如图 3.28 第 3 行所示, C 语言的复合语句以左大括号开始, 接着是若干条的声明, 后面再跟着若干条语句, 最后以右大括号结束。在 C 语言中, 一旦我们遇到复合语句的左括号, 代表这是一个新的作用域的开始, 我们可以在这个新的作用域里声明一些新的局部变量, 第 15 行的“Level++;”记录了作用域深度的变化, 而第 37 行的“Level--;”则表示我们在遇到复合语句的右大括号时, 就离开了当前作用域。C 程序员通过声明, 给编译器提供了类型信息, C 编译器会用这些信息来创建类型结构。图 3.28 第 16 行构造了一个 struct astCompoundStatement 的语法树结点。图 3.28 第 18 至 26 行通过调用 ParseDeclaration() 函数实现了对若干条声明的分析, 而第 27 至 34 行的代码通过依次调用 ParseStatement() 完成了对各个语句的分析, 若在第 31 行发现当前 token 已是右大括号, 则可跳出 while 循环。在第 19 行和第 33 行中, 集合 First\_Declaration 和 Fisrt\_Statement 正是我们第 1 章介绍过的“首符集”的概念, 我们可以根据 C 标准文法中与“声明和语句”相关的产生式, 得到它们的首符集, 如图 3.29 所示。

```

1 // tokens that may be first token in a declaration
2 #define FIRST_DECLARATION \
3     TK_AUTO,    TK_EXTERN,    TK_REGISTER,    TK_STATIC, \
4     TK_TYPEDEF,  TK_CONST,    TK_VOLATILE,   TK_SIGNED, \
5     TK_UNSIGNED, TK_SHORT,   TK_LONG,      TK_CHAR, \
6     TK_INT,     TK_INT64,   TK_FLOAT,    TK_DOUBLE, \
7     TK_ENUM,    TK_STRUCT,  TK_UNION,    TK_VOID,  TK_ID
8
9 // fisrt token of an expression
10 #define FIRST_EXPRESSION \
11     TK_SIZEOF,  TK_ID,     TK_INCONST,  TK_UINTCONST, TK_LONGCONST, \
12     TK ULONGCONST, TK_LLONGCONST, TK_ULLONGCONST, TK_FLOATCONST, \
13     TK_DOUBLECONST, TK_LDOUBLECONST, TK_STRING,   TK_WIDESTRING, \
14     TK_BITAND,   TK_ADD,    TK_SUB,     TK_MUL,    TK_INC, \
15     TK_DEC,     TK_NOT,    TK_COMP,    TK_LPAREN
16

```

```

17 // first token of statement
18 #define FIRST_STATEMENT \
19     TK_BREAK, TK_CASE, TK_CONTINUE, TK_DEFAULT, TK_DO, \
20     TK_FOR, TK_GOTO, TK_IF, TK_LBRACE, TK_RETURN, TK_SWITCH, \
21     TK_WHILE, TK_SEMICOLON, FIRST_EXPRESSION

```

图 3.29 首符集

图 3.29 第 2 至 7 行列出了“声明”的首符集，而第 10 至 15 行列出了“表达式”的首符集，第 18 至 21 行列出了“语句”的首符集。因为表达式后面跟上一个分号，就可以构成一个表达式语句，所以“表达式首符集”是“语句首符集”的子集，如第 21 行末尾的 FIRST\_EXPRESSION 所示。我们可以发现，终结符 TK\_ID 可以是声明的首符，也可以是表达式的首符，当然也可以是语句的首符。但按 C 的文法，充当声明首符的标志符 TK\_ID 应该是个类型名，在图 3.28 第 22 行的 if 语句就用来判断当前 token 是否只是一个普通的标志符，其中的函数 IsTypeName() 用于“用于判断标志符是否为类型名”，例如图 3.28 第 20 至 21 行注释中的函数名 f 就不是一个类型名，这表明复合语句中的声明部分已经结束，我们可以跳入图 3.28 第 27 行对若干条语句进行分析。图 3.30 给出了一个复合语句的例子及其对应的语法树。

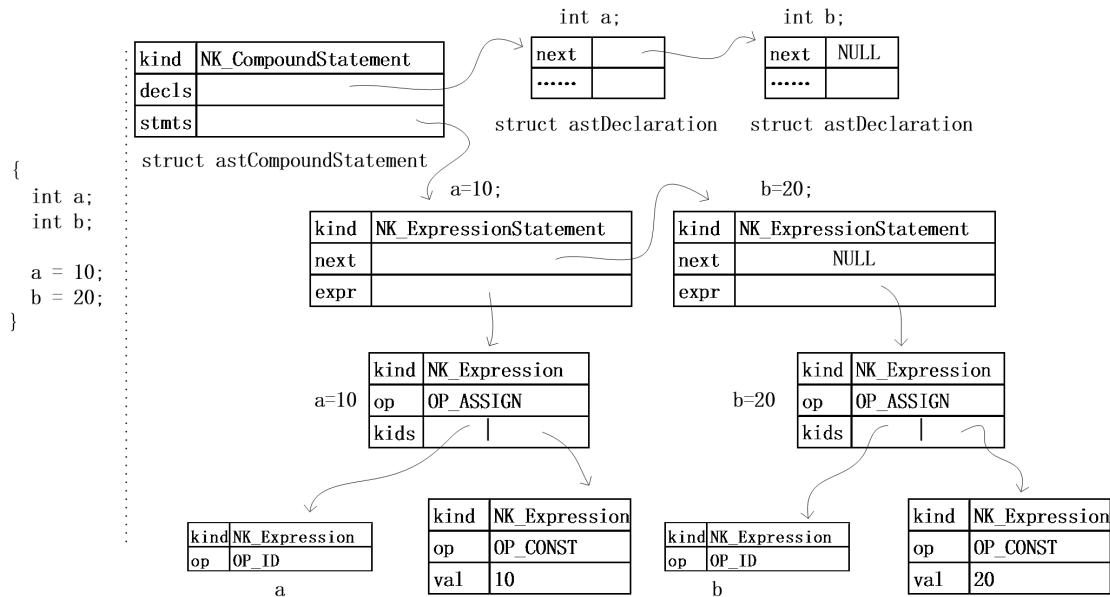


图 3.30 复合语句对应的语法树

在 3.1 节，我们介绍了通过 `typedef` 建立的类型名可能会被另作他用，例如，以下标志符 `ddd` 在函数 `g` 中就是一个局部变量名，而不是类型别名。C 语言的函数体实际上就是一个复合语句。当离开复合语句，遇到“`ddd d;`”时，我们又要把 `ddd` 当作类型名。

```

typedef int ddd;
void g(){
    int ddd;
}
ddd d;

```

我们会在图 3.28 第 36 行调用 `PostCheckTypedef()` 函数来完成这些工作，其代码如图 3.31 所示。在向量 `TypedefNames` 和 `OverloadNames` 中存放的是若干个 `struct tdname` 对象的指针，在即将离开函数 `g` 时，我们通过图 3.31 第 6 至 10 行的循环，把在当前作用域中由 `typedef` 定义的类型名设为失效，只要在第 8 行把 `tn->id` 置为 `NULL` 即可，没有必要从向量 `TypedefNames` 中真正删除元素。接下来，我们能过第 12 至 18 行的循环，检查一下当前作用域中是否有把类型名另作他用，如果有则通过第 14 行把 `overload` 标志清 0。图 3.31 第 15

至 17 行用于统计在向量 OverloadNames 中仍然存在重载的类型名的个数。如果向量 OverloadNames 中已经没有重载的类型名，则通过第 21 行把这个向量清空。由于我们会在图 3.31 第 14 行置 overload 为 0，这样，在离开上述函数 g 后再次遇到“ddd d;”中的 ddd 时，我们就会将其当作一个类型名。

```

1 void PostCheckTypedef(void)
2 {
3     TDName tn;
4     int overloadCount = 0;
5     // make typedefNames in current scope invalid
6     FOR_EACH_ITEM(TDName, tn, TypedefNames)
7         if(Level == tn->level){
8             tn->id = NULL;
9         }
10    ENDFOR
11    // restore the overloaded typedefNames
12    FOR_EACH_ITEM(TDName, tn, OverloadNames)
13        if(Level <= (tn->overloadLevel)){
14            tn->overload = 0;
15        }else if(tn->overload != 0){
16            overloadCount++;
17        }
18    ENDFOR
19    // empty the vector if no overloaded names in it
20    if(overloadCount == 0){
21        OverloadNames->len = 0;
22    }
23 }
```

图 3.31 PostCheckTypedef()

当通过 `typedef` 定义了一个类型名时，我们需要把这个类型名加入到向量 `TypedefNames` 中。在我们声明一个形式参数、局部变量和全局变量时，就需要检查一下变量名是否跟已有的由 `typedef` 得来的类型名重名，如果有重名则把这个信息记录到向量 `OverloadNames` 中。如图 3.32 所示的函数 `CheckTypedefName()` 完成了这个工作。

```

1 static void CheckTypedefName(int sclass, char *id)
2 {
3     Vector v;
4     TDName tn;
5
6     if (id == NULL)
7         return;
8
9     v = TypedefNames;
10    if (sclass == TK_TYPEDEF){
11        FOR_EACH_ITEM(TDName, tn, v)
12            if(tn->id == id && Level == tn->level){
13                return;
14            }
15    ENDFOR
16    // add one typedefName
17    ALLOC(tn);
18    tn->id = id;
19    tn->level = Level;
```

```

20     tn->overload = 0;
21     INSERT_ITEM(v, tn);
22 }else{
23     FOR_EACH_ITEM(TDName, tn, v)
24     if (tn->id == id && Level > tn->level) {
25         tn->overload = 1;
26         tn->overloadLevel = Level;
27         INSERT_ITEM(OverloadNames, tn);
28     }
29 }
30 ENDFOR
31 }
32 }
```

图 3.32 CheckTypedefName()

图 3.32 第 11 至 15 行先检查一下在当前作用域中是否已经有定义过同名的类型名，如果有则直接返回；否则通过第 17 至 20 行的代码往 `typedefNames` 中添加一个新的类型名。在定义一个形参、局部变量或者全局变量之后，我们通过图 3.32 第 24 行的 `if` 语句来检查一下在更浅层的作用域中是否有同名的 `typedef` 类型名，如果有则在第 25 行置标志 `tn->overload` 为 1，并在第 26 行把当前所处的作用域深度记录到 `overloadLevel` 中，然后通过第 27 行把相关信息加入到 `overloadNames` 向量中。函数 `CheckTypedefName()`会在对“声明”进行语法分析时被调用，我们会在下一节进行讨论。

## 3.3 C 语言的外部声明

### 3.3.1 声明和函数定义

在分析完表达式和语句之后，对于 C 语言的标准文法，我们就已经差不多已经理解了一半，另外的文法基本上都是关于声明的，与声明相关的代码在 `ucldecl.c` 中。在不少程序设计的书籍中，对声明与定义是有严格区分的，当编译器遇到定义时，需要为之分配内存；而遇到声明时，则不必为之分配内存，因为该声明对应的内存可能已在其他地方进行过分配。例如，以下代码中，“`extern int a;`”被当作对变量 `a` 的声明；而“`int b = 5;`”则被当作定义，需要为变量 `b` 分配存储空间，并作相应初始化；“`void f(void);`”被当作对函数的声明，不需要占任何代码区空间；而接下来的是函数 `g` 的定义，需要在代码区中占用一片内存来存放其代码。

```

extern int a;
int b = 5;
void f(void);
void g(void){
    //...
}
```

不过，从 C 标准文法的角度出发，上述“需要占内存的定义”或者“不占内存的声明”统统是被当作“外部声明（external-declaration）”来处理的。从 C 语言的语义出发，上述外部声明才有“是否要占内存”之分。换句话说，在语法分析阶段，我们暂时只在乎 C 程序在形式上是否符合 C 标准文法，我们在这个阶段只管“形”，而不管“含义”或“内容”。在语法分析阶段，一个预处理后的 C 文件就对应一个翻译单元（translation-unit），它由若干个外部声明构成，如图 3.33 第 2 至 4 行的产生式所示。

```
1  /**
```

```

2  * translation-unit:
3  *      external-declaration
4  *      translation-unit external-declaration
5  */
6 AstTranslationUnit ParseTranslationUnit(char *filename) {
7     AstTranslationUnit transUnit;
8     AstNode *tail;
9
10    ReadSourceFile(filename);
11    TokenCoord.filename = filename;
12    TokenCoord.line = TokenCoord.col = TokenCoord.ppline = 1;
13    TypedefNames = CreateVector(8);
14    OverloadNames = CreateVector(8);
15    CREATE_AST_NODE(transUnit, TranslationUnit);
16    tail = &transUnit->extDecls;
17    NEXT_TOKEN;
18    while (CurrentToken != TK_END) {
19        if (CurrentToken == TK_SEMICOLON) {
20            NEXT_TOKEN;
21            continue;
22        }
23        *tail = ParseExternalDeclaration();
24        tail = &(*tail)->next;
25        SkipTo(FIRST_ExternalDeclaration, "the beginning of external declaration");
26    }
27    CloseSourceFile();
28    return transUnit;
29 }

```

图 3.33 ParseTranslationUnit()

在图 3.33 第 10 行调用的函数 `ReadSourceFile()` 用于读入预处理后的 C 文件，第 11 和 12 行用于初始化当前读头的坐标位置，第 13 行创建的向量 `TypedefNames` 用于存放通过 `typedef` 命名的类型名，而接下来的 `OverloadNames` 则用于记录“另作他用”的 `typedef` 类型名，我们在前文中已举例介绍过这两个向量的用法。第 15 行创建了一个 `kind` 域为 `NK_TranslationUnit` 的语法树结点。一个翻译单元是由若干个外部声明构成的，对外部声明的分析主要是通过第 18 行的 `while` 循环和第 23 行调用的函数 `ParseExternalDeclaration()` 来实现的。引入第 19 至 22 行的代码是为了允许形如“`;`”这样的空语句，而第 16 行、23 行和 24 行的代码则用于把各个外部声明所对应的语法子树链接到一起。执行到第 27 行时，我们就已经完成了对一个预处理后的 C 文件的语法分析工作，通过调用函数 `CloseSourceFile()` 来关闭相应文件即可。图 3.34 给出了一个预处理后的 C 文件及其对应的语法树示意图。

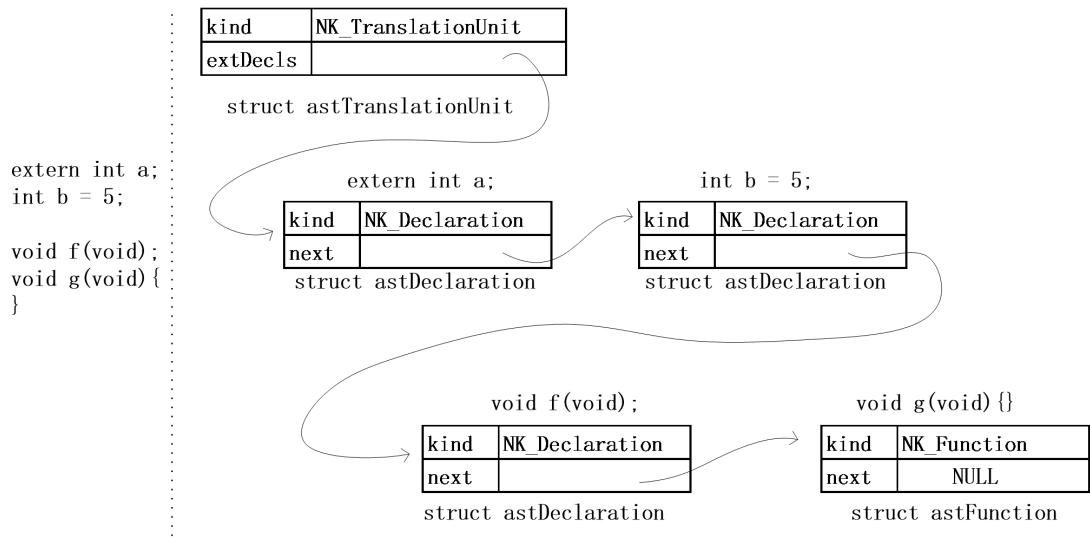


图 3.34 TranslationUnit 对应的语法树

通过在图 3.33 第 18 行 `while()` 循环内调用函数 `ParseExternalDeclaration()`, 我们在图 3.34 中依次创建了 `kind` 域为“`NK_Declaration` 或 `NK_Function`”的 4 棵语法子树, 其中 `NK_Function` 对应的是 `struct astFunction` 对象, 描述了函数定义 `g` 的相关信息; 而 `NK_Declaration` 对应的是 `struct astDeclaration` 对象, 描述了变量声明 `a`、变量定义 `b` 和函数声明 `f`。而用于分析“外部声明”的函数 `ParseExternalDeclaration()` 如图 3.35 所示。

```

1 /**
2 *  external-declaration:
3 *      function-definition
4 *      declaration
5 *
6 *  function-definition:
7 *      declaration-specifiers declarator [declaration-list] compound-statement
8 *
9 *  declaration:
10 *      declaration-specifiers [init-declarator-list] ;
11 *
12 *  declaration-list:
13 *      declaration
14 *      declaration-list declaration
15 */
16 static AstNode ParseExternalDeclaration(void) {
17     AstDeclaration decl = NULL;
18     AstInitDeclarator initDec = NULL;
19     AstFunctionDeclarator fdec;
20
21     decl = ParseCommonHeader();
22     initDec = (AstInitDeclarator) decl->initDecls;
23     if (decl->specs->stgClasses != NULL &&
24         ((AstToken) decl->specs->stgClasses)->token == TK_TYPEDEF) {
25         goto not_func;
26     }
27     fdec = GetFunctionDeclarator(initDec);
28 //略

```

29 }

图 3.35 ParseExternalDeclaration()

图 3.35 第 2 至 4 行告诉我们，外部声明由“函数定义”或“声明”构成。由第 6 至 10 行，我们可以发现，这两个候选式有完全一样的前缀 declaration-specifiers，即“声明说明符”。以下代码中，static、const 和 int 都被称为声明说明符。语法上，声明说明符可分为“存储类说明符”、“类型限定符”和“类型说明符”这 3 类。

```
//对应图 3.35 第 9 行的 Declaration
static const int a1 = 3, * a2, a3[5], a4(void);
//对应图 3.35 第 6 行的 FunctionDefinition
static void a5(void) {
}
```

(1) 关键字 static 被称为存储类说明符 (storage-class-specifier)，与之类似的还有 auto、register 和 extern，这些说明符暗示了变量的存储位置，例如 auto 表明变量是“自动变量”，即局部变量对应的栈空间是在函数返回后自动回收的，不需要由 C 程序员负责。不过，在编写 C 程序中，我们没有必要显式地把局部变量声明为 auto；而关键字 register 则建议 C 编译器尽量把局部变量或形参存放到寄存器中，只是建议而已，实际上 C 程序员也很少需要显式地使用关键字 register。现代的 C 编译器在优化方面下足了功夫，总是想方设法地想重用寄存器中的值，因此，保留 register 关键字更多的是为了兼容旧代码。

(2) 关键字 const 则被称为类型限定符 (type-qualifier)，与之地位相似的还有 volatile。

(3) 关键字 int 则被称为类型说明符 (type-specifier)，与之类似的还有 void、char、short、float、double 和结构体说明符等。通过结构体说明符，C 程序员可以创建自定义的类型。由此我们也能看到，在语法上，结构体等用户自定义类型与 C 语言内置的 int 类型的地位是相当的。

而跟在声明说明符 “static const int” 之后的，则是由若干个“可带初值的声明符构成的声明符列表 (init-declarator-list)”，例如上述的 “a1=3”，“\*a2”、“a3[5]” 和 “a4(void)”。其中的 “a1=3” 是带初值的声明符。实际上，我们在第 1.3 节时就介绍过声明符的概念。在形如 “\*a2”的声明符中，我们声明了指针类型，在 UCC 内部，“\*a2” 会对应一个 kind 域为 NK\_PointerDeclarator 的语法树结点；而在形如 “a3[5]” 的声明符中，我们声明了数组类型，在语法树上，“a3[5]” 会对应一个 kind 域为 NK\_ArrayDeclarator 的结点；而在形如 “a4(void)” 的声明符中，我们声明了函数类型，与之对应的是一个 kind 域为 NK\_FunctionDeclarator 的语法树结点。结合这个简单的例子，我们介绍了 C 标准文法中出现的声明说明符和声明符的含义。由以上分析可以发现，C 语言的类型信息正是由声明说明符 (declaration-specifiers) 和声明符 (declarator) 这两者构成。C 语言中的 int 等基本类型、结构体 struct、联合体 union 和枚举 enum 会出现在声明说明符 (declaration-specifiers) 中；而数组、指针和函数类型则由声明符 (declarator) 来生成。

存储类说明符、类型限定符和类型说明符等概念一开始接触时，似乎不太容易记住，但至少我们可以有一个大的概念，那就是这些都是声明说明符，然后我们记住有形如 “static const int” 这样的声明说明符就可以了，接下来我们很容易联想到与 static 地位相同的有哪些，与 const 地位相当的有哪些，而与 int 类似的又有哪些。有了这些概念后，我们再来看图 3.35 第 6 至 7 行，可以发现在 function-definition 中出现的只是 declarator，而非第 10 行的 init-declarator-list，而 declarator 只描述了一个不带初值的声明符，通过 init-declarator-list 我们才能描述多个“可带初值的声明符”。因此，以下函数定义是非法的，其中出现了 a6 和 a7 两个声明符，且 a6 还进行了非法的初始化。

```
void a6(void)=3, a7(void) {
```

} 由于声明 (declaration) 和函数定义 (function-definition) 有共同的前缀，UCC 编译器

为了简化语法分析的工作，就把出现在 function-definition 中的 declarator 也当作 init-declarator-list 来处理，这当然不够精确，但我们会在后续阶段进行修正。如此处理后，这两者的公共前缀就如下所示，下标 opt 是 optional 的缩写，表示“可选的”。

`DeclarationSpecifiersopt InitDeclaratorListopt`

图 3.35 第 21 行通过调用 `ParseCommonHeader()` 函数，完成了这部分公共前缀的分析。

此时我们会误把“void a6(void)=3, a7(void)”当作合法的输入。第 23 行用来判断一下在公共前缀的存储类说明符中，是否出现过 `typedef` 关键字。在语法上，`static` 和 `typedef` 的地位相当，C 标准文法把 `typedef` 关键字也当作一种存储类说明符，如下代码所示；当然在语义上，`typedef` 和 `static` 是风马牛不相及。按 C 的文法，如果遇到了 `typedef` 关键字，我们可以百分之百确定这不是一个函数定义。

```
typedef const int ABC;
static const int a8;
```

对于“`static const int a1 = 3, * a2, a3[5], a4(void);`”而言，我们期望通过函数调用 `ParseCommonHeader()` 能为之构造一棵形如图 3.36 的语法树。

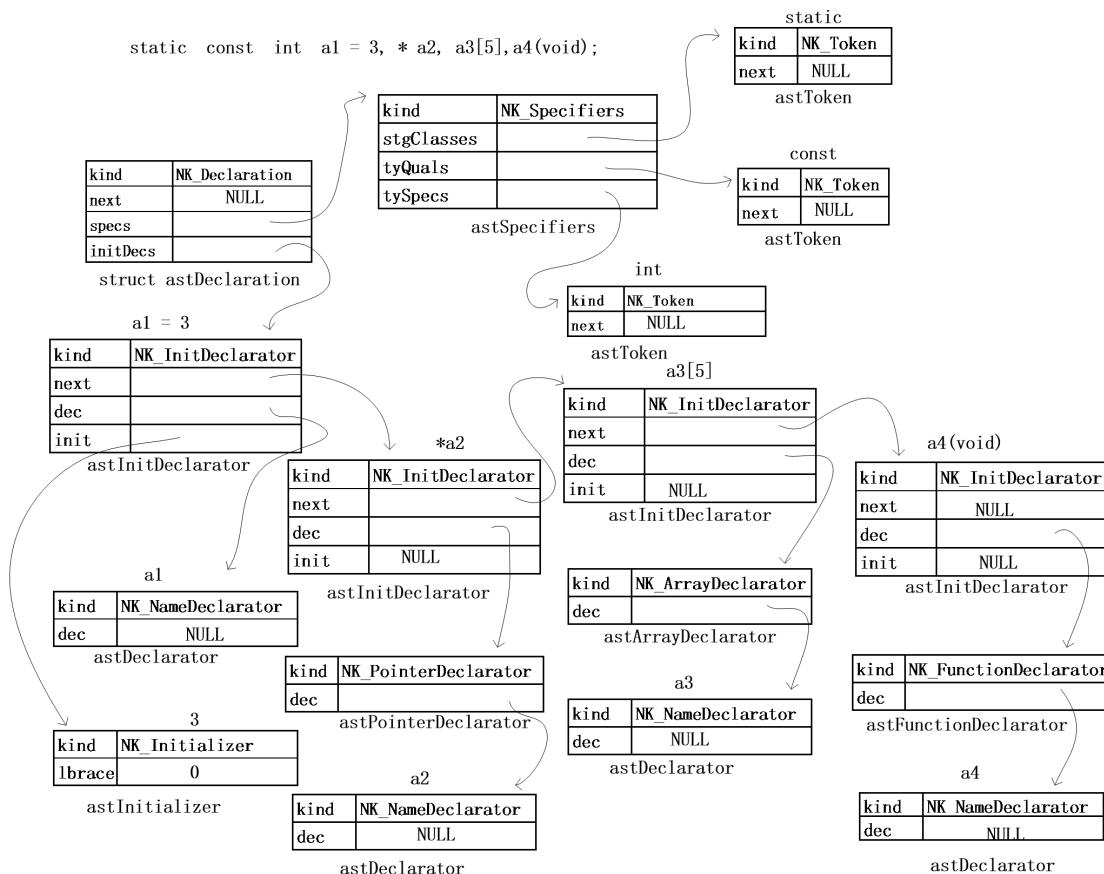


图 3.36 声明 Declaration 所对应的语法树

在图 3.36 中，在 kind 域为 `NK_Declaration` 的语法树结点记录了上述声明的相关信息，主要有这么两部分，一部分由是由 `specs` 域所指向的声明说明符相关信息，另一部分是 `initDecls` 所指向的初始化声明符链表，该链表由 4 个 kind 域为 `NK_InitDeclarator` 的语法树结点构成。在 kind 为 `NK_InitDeclarator` 的结点上，由 `dec` 域指向其声明符，而由 `init` 域指向其初值，若不存在初值，则 `init` 域为 `NULL`。

如果我们要分析的正是一个声明（declaration），则通过在图 3.35 第 21 行调用的函数 `ParseCommonHeader()`，我们就构成了形如图 3.36 的语法树，就基本完成了对声明的分析工作。但如果要分析的是一个函数定义（function-definition），则需要作进一步的修正。首先我

们需要检查一下所构造的语法树上，是否存在 kind 域为 NK\_FunctionDeclarator 的结点，如图 3.36 所示，图 3.35 第 27 行调用函数 GetFunctionDeclarator(initDec)完成了这个检查。该函数对应的代码如图 3.37 所示，第 2 行的参数 initDec 指向图 3.36 中由若干个 NK\_InitDeclarator 结点构成的链表。

```

1 static AstFunctionDeclarator GetFunctionDeclarator(
2                         AstInitDeclarator initDec) {
3     AstDeclarator dec;
4     // (1) initDec->next != NULL
5     //      int abc, g(int a,int b) { }      is illegal.
6     // (2) initDec->init != NULL
7     //      int g(int a, int b) = 30;        is illegal.
8     if (initDec == NULL || initDec->next != NULL || initDec->init != NULL) {
9         return NULL;
10    }
11    dec = initDec->dec;
12    /*****
13     int (* f(void)) (void){
14         return 0;
15     }
16     对应的链表为:
17     NK_FunctionDeclarator->NK_PointerDeclarator
18         -> NK_FunctionDeclarator->NK_NameDeclarator
19     *****/
20    while (dec ) {
21        if(dec->kind == NK_FunctionDeclarator
22            && dec->dec && dec->dec->kind == NK_NameDeclarator) {
23            break;
24        }
25        dec = dec->dec;
26    }
27    return (AstFunctionDeclarator)dec;
28 }
```

图 3.37 GetFunctionDeclarator()

由于函数定义的产生式中只允许出现一个声明符，且不带初值，因此在函数定义对应的 NK\_InitDeclarator 链表中只能包含一个结点，图 3.37 第 8 行的 if 语句按这个要求进行了判断，第 4 至 7 行注释中的例子对第 8 行的“initDec->next != NULL”和“initDec->init != NULL”做了说明。接着我们想沿着该 NK\_InitDeclarator 结点的 dec 域去查找，看看有没有 kind 域为 NK\_FunctionDeclarator 的结点，且该结点的后继应该是一个 NK\_NameDeclarator 结点，如图 3.36 所示。图 3.37 第 20 至 26 行的 while 循环实现了这个查找。第 12 至 19 行的注释举了一个具体的例子，用来说明为什么我们仅仅去查找 kind 域为 NK\_FuctionDeclarator 的结点是不够的，而是要再判断一下该结点的后继结点是否对应一个函数名，即 NK\_NameDeclarator 结点。如果我们找不到满足条件的 NK\_NameDeclarator 结点，则返回 NULL，这说明我们刚刚遇到的不是一个函数定义；否则返回该结点的地址，但这只能说明我们可能找到了一个函数定义，该结点也可能对应的是一个函数声明。因此，还需要再做进一步分析，看看接下来的 token 是不是左大括号，如果是则说明我们遇到了函数体，此时可以确定我们正在分析的是一个函数定义，而非函数声明。图 3.35 只给出了函数 ParseExternalDeclaration() 的部分代码，图 3.38 给出了 ParseExternalDeclaration() 的其余代码，这部分代码主要是用于处理函数定义的。

```
1 if (fdec != NULL) {
```

```

2     AstFunction func;
3     AstNode *tail;
4     if (CurrentToken == TK_SEMICOLON) {
5         NEXT_TOKEN;
6         if (CurrentToken != TK_LBRACE)
7             return (AstNode)decl;
8         Error(&decl->coord, "maybe you accidentally add the ;");
9     }else if(fdec->paramTyList && CurrentToken!= TK_LBRACE) {
10        goto not_func;
11    }
12    CREATE_AST_NODE(func, Function);
13    func->coord = decl->coord;      func->specs = decl->specs;
14    func->dec = initDec->dec;      func->fdec = fdec;
15    Level++;
16    if (func->fdec->paramTyList) {
17        AstNode p = func->fdec->paramTyList->paramDecls;
18        while (p) {
19            CheckTypedefName(0,GetOutermostID(((AstParameterDeclaration)p)->dec));
20            p = p->next;
21        }
22    }
23    tail = &func->decls;
24    while (CurrentTokenIn(FIRST_Declaration)) {
25        *tail = (AstNode)ParseDeclaration();
26        tail = &(*tail)->next;
27    }
28    Level--;
29    func->stmt = ParseCompoundStatement();
30    return (AstNode)func;
31 }
32 not_func:
33 if(!decl->specs->stgClasses && !decl->specs->tyQuals && !decl->specs->tySpecs) {
34     Warning(&decl->coord,"declaration specifier missing, defaulting to 'int'");
35 }
36 Expect(TK_SEMICOLON);
37 PreCheckTypedef(decl);
38 return (AstNode)decl;
39 }

```

图 3.38 FunctionDefinition

对照着如下所示的函数定义的产生式，我们可以较好地理解图 3.38 中的代码。通过 ParseCommonHeader() 和 GetFunctionDeclarator() 这两个函数，我们实际上已经分析了产生式中的“declaration-specifiers declarator”。

function-definition:

declaration-specifiers declarator [declaration-list] compound-statement

接下来的是一个可选的声明列表（declaration-list），这部分主要是用于如下所示的旧式风格函数定义，其中的“int a;double b,c;”就是个可选的声明列表，图 3.38 的第 23 至 27 行完成了对此的分析。不论是新式风格，还是旧式风格的函数定义，其函数体都是一个以左大括号开头的复合语句，第 29 行调用 ParseCompoundStatement() 完成了对函数体的分析。

```

void fo(a,b,c) int a;double b,c{
}

```

我们在 2.4 节介绍 C 语言的类型系统时介绍过，尽量远离旧式风格的 C 函数，但是 C

编译器需要兼容已有的代码，所以还要不厌其烦地面对这些历史包袱。如果是以下的新式风格函数定义，通过 ParseCommonHeader() 我们就已经分析了其中的 “`int * fn(int aa,double bb,double cc)`”，接下来的就是应是左括号开头的函数体了。由于通过 `typedef` 我们可以命名一些类型，但是这些类型名可能会跟函数 `fn` 中的形参 `aa`、`bb` 或 `cc` 重名，我们需要把这些信息记录在向量 `TypedefNames` 和 `OverloadNames` 中，图 3.38 第 16 至 22 行的 while 循环通过调用函数 `CheckTypedefName()` 完成了这个工作。而图 3.38 第 12 行创建的 `struct astFunction` 对象则用于记录与函数定义相关的所有信息。

```
int * fn(int aa,double bb,double cc) {
    return NULL;
}
```

图 3.39 给出了函数 `fn` 对应的语法树，结合这棵语法树，用于对函数定义进行语法分析的图 3.38 就会一目了然。

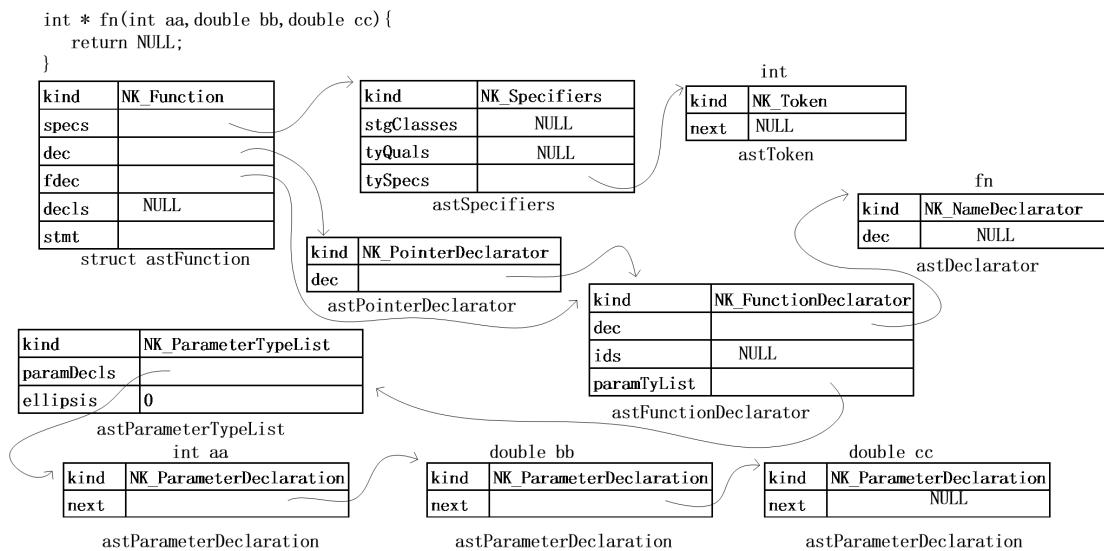


图 3.39 函数定义所对应的语法树

在图 3.39 的 kind 域为 `NK_Function` 的结构体 `struct astFunction` 对象中，记录了与函数定义 `fn()` 有关的所有信息，该对象的 `specs` 用来存放声明说明符，而 `dec` 域存放了如下所示的单链表，而 `fdec` 域则指向该链表中 kind 域为 `NK_FunctionDeclarator` 且后继结点为 `NK_NameDeclarator` 的结点。

`astPointerDeclarator --> astFunctionDeclarator --> astDeclarator(fn)`

如果逆序遍历（按从右至左）这条单链表，结合在图 3.39 的 `astSpecifiers` 对象中存放的类型说明符 `int`，我们依次可以得到这样的类型信息：

- (1) 访问 `astDeclarator` 对象得到函数名 `fn`
- (2) 访问 `astFunctionDeclarator` 对象得到： `fn is a function`
- (3) 访问 `astPointerDeclarator` 对象得到： `fn is a function which returns pointer to`
- (4) 访问 `astSpecifiers` 对象得到： `fn is a function which returns pointer to int`

实际上，我们在第 1.3 节讨论 `ucc\examples\sc` 的声明符时，也遇到过类似的单链表。UCC 在后续阶段会对这条单链表进行处理，最终通过函数 `DeriveType()` 推导出 `fn` 的类型信息，并记录在“第 2.4 节 C 语言类型系统”中介绍过的 `struct functionType` 对象中。而函数 `DeriveType()` 的代码，我们会在语义检查时再进行讨论。而函数 `fn` 的形参列表，则存放在图 3.39 中的 `astParameterTypeList` 对象中，通过 `astFunctionDeclarator` 对象的 `paramTyList` 域可找到该对象。而在 kind 域为 `NK_ParameterTypeList` 的语法树结点中，其 `ellipsis` 域为 0，表示 `fn` 不是一个变参函数；其 `paramDecls` 域则指向了由若干个 `astParameterDeclaration` 对象构

成的链表，每个 `astParameterDeclaration` 对象可用于记录一个形参信息，如图 3.39 所示。而 `struct astDeclaration` 用于描述“局部变量或全局变量声明”，可以发现这两个结构体几乎是一样的，如下所示：

```
struct astDeclaration{
    AST_NODE_COMMON
    AstSpecifiers specs;
    AstNode initDecls;
};

typedef struct astParameterDeclaration{
    AST_NODE_COMMON
    AstSpecifiers specs;
    AstDeclarator dec;
} *AstParameterDeclaration;
```

它们的差别在于 C 语言的形参声明不可以有初值，例如以下函数定义 `fe` 是非法的，而局部变量 `b` 和全局变量 `c` 的声明可以有初值。带常量初值的形参，在 C++ 中被当作默认参数，但 C 语言视之为非法。所以，合不合法，还是制定规则的人说了算。

```
void fe(int a =3){
    int b = 4;
}
int c = 5;
```

当发现通过 `ParseCommonHeader()` 分析的只是一个声明，而不是函数定义时，我们会执行图 3.38 第 32 行的代码，此处要做的主要工作是检查一下在声明中，各个声明符的名称是否与 `typedef` 定义的类型名重名，这通过调用函数 `PreCheckTypedef()` 来实现，其对应的代码如图 3.40 所示，结合图 3.36 所示的数据结构，我们不难理解第 4 至第 8 行的代码。

```
1 static void PreCheckTypedef(AstDeclaration decl) {
2     AstNode p;
3     int sclass = 0;
4     if (decl->specs->stgClasses != NULL) {
5         sclass = ((AstToken) decl->specs->stgClasses)->token;
6     }
7
8     p = decl->initDecls;
9     while (p != NULL) {
10        CheckTypedefName(sclass, GetOutermostID(((AstInitDeclarator)p)->dec));
11        p = p->next;
12    }
13 }
```

图 3.40 PreCheckTypedef()

对于“`static const int a1 = 3, * a2, a3[5], a4(void);`”而言，通过在图 3.40 第 10 行调用 `GetOutermostID()` 函数，我们可以获得各个声明符的名称，即此处的 `a1`、`a2`、`a3` 和 `a4`。再通过调用 `CheckTypedefName()` 函数检查一下这些名称是否与 `typedef` 定义的类型名重名，`CheckTypedefName()` 的代码已在 3.2 节讨论过。

### 3.3.2 与声明有关的几个非终结符

在上一小节中，我们讨论了 C 标准文法的“外部声明”，介绍了“声明”和“函数定义”这两类外部声明，分别举了以下代码作为例子，在图 3.36 和图 3.39 中给出了这两者对应的语法树。

```
//声明
```

```

static const int a1 = 3, * a2, a3[5], a4(void);
//函数定义
int * fn(int aa,double bb,double cc){
    return NULL;
}

```

我们需要对 ucl\decl.c 的其他代码进行分析，才能更好理解图 3.36 和图 3.39 的语法树是如何构造出来的。这里可能有个问题，程序的原作者在编程时，在其脑海中就已经有其要构造的语法树的整体轮廓；但是，程序的阅读者在缺乏文档的情况下，只有读完大部分代码，做过分析后才能重构出这棵语法树的概貌。另一个问题是，C 标准文法刻画的是 C 语言这样的无穷集合，不同的 C 程序对应不同的语法树，而在书中，我们只能选取一些如上所示的简单而又有一定代表性的代码来举例，在还没有介绍完 ucl\decl.c 的所有函数时，就提前给出形如图 3.36 和图 3.39 这样的语法树预览图，之后结合语法树，就可以更好地阅读和分析 ucl\decl.c 中的代码。这有点类似于“玩拼图”，在开始动手拼图时，如果能提前知道最终要完成的图案，那应该会更容易些。

通过上一小节的讨论，我们知道对于“声明”和“函数定义”这两类“外部声明”的分析主要是由 ParseCommonHeader() 来完成，这个所谓的公共前缀（CommonHeader）实际上就是“声明”的候选式；而对于“函数定义”来说，我们先把其前缀“误当”成和“声明”一样。至于两者不一样的地方，我们已经在讨论 ParseExternalDeclaration() 时进行过修正。

在 ParseCommonHeader() 函数的基础上，来实现对“声明”的分析，就是易如反掌的事情，如图 3.41 所示。由于全局变量的声明属于外部声明，已在函数 ParseExternalDeclaration() 中完成分析，所以图 3.41 中的 ParseDeclaration() 函数主要用于对局部声明进行语法分析，会在函数 ParseCompoundStatement() 中被调用。

```

1  ****
2 declaration
3   declaration-specifiers [init-declarator-list]
4   ****
5 AstDeclaration ParseDeclaration(void) {
6   AstDeclaration decl;
7
8   decl = ParseCommonHeader();
9   Expect(TK_SEMICOLON);
10  PreCheckTypedef(decl);
11  return decl;
12 }

```

图 3.41 ParseDeclaration()

在对“声明”的相关代码做进一步讨论前，让我们再次强调一下，图 3.41 第 2 至 3 行的产生式中的声明说明符（declaration-specifiers）对应的是形如“static const int”这样的字符串，对 int 和 double 等基本类型的处理相对比较简单，这里我们要重点讨论的是“结构或联合说明符（struct-or-union-specifier）”。而可带初值的声明符对应的是形如“a1 = 3”这样的字符串。我们在第 1 章讨论 ucc\examples\sc 时就已介绍过声明符的概念，在标准 C 中，可为声明符加上初值，构成 init-declarator，而多个 init-declarator 则构成上述产生式中的 init-declarator-list。前文所述的 a1, \* a2, a3[5] 和 a4(void) 都是 C 语言中的声明符。声明符中包含了数组、指针和函数等类型信息，而声明说明符中则包含了“int 和 double 等基本类型”、结构体类型、联合体类型和枚举类型等信息。这两者结合在一起，就构成了 C 语言的声明。C 程序员通过编写一个声明来构造一个类型表达式，C 编译器利用这些类型表达式来构建类型系统。

如果去掉上述声明符中的形如 a1、a2、a3 和 a4 这样的标志符，剩下的字符串就对应 C

标准文法中抽象声明符（abstract-declarator）的概念。引入“抽象声明符”的原因在于，在C语言的 sizeof 表达式、函数声明中的形参和强制类型转换中，我们关注的是类型信息，并不是去声明一个变量，此时并不需要声明符中的标志符，如下所示。

```
sizeof(int * [4]);           // sizeof(TypeName)
void fdecl(int (*)[4]); // parameter-declaration in function declaration
(int *) ptr;                // (TypeName) UnaryExpression
```

由于引入了抽象声明符来刻画函数声明中的形参，所以 C 标准文法就不能延用之前的非终结符 declaration 来描述形参声明，需要另外创建一个被称为“形式参数声明（parameter-declaration）”的非终结符来表达，其产生式如下所示。由这个产生式可知，“void f(int);”和“void f(int a);”都是合法的声明，即在形参声明中，跟在声明说明符之后的可以是声明符，也可以是抽象声明符。

```
parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator
```

在标准 C 文法中，还引入了“结构体成员声明符（struct-declarator）”的概念，主要是用于描述如下所示的结构体位域成员 offset 和 number。

```
struct Entry{
    int offset:12;
    int number:20;
    int date;
}
```

结构体成员声明符对应的产生式如下所示，对于结构体定义中的非位域成员，例如上述结构体的 date 域，我们完全可以用之前介绍过的声明符来描述。因此，struct-declarator 有两个候选式，如下所示：

```
struct-declarator:
    declarator
    declaratoropt : constant-expression
```

由于结构体只是用来描述数据结构的，在结构体内部，我们不需要 static、register 和 auto 等跟存储有关的声明说明符。这些信息，只有在需要为整个结构体对象分配存储位置时才需要说明。如果允许结构体内部的声明中出现 register 和 auto 等，就可能出现如下情况，data1 被建议放在寄存器中，而 data2 要放于内存栈中，这明显矛盾。

```
struct Data{
    register int data1;    //StructDeclaration      结构体成员声明
    auto int data2;        //StructDeclaration      结构体成员声明
};
```

最终，在 C 标准文法中，与结构体成员声明相关的产生式如下所示。由前述的形如“static const int”的声明说明符中删去 static，剩下的“const int”就对应如下的“说明符限定符列表（specifier-qualifier-list）”。简单来说，看到非终结符 specifier-qualifier-list，我们脑子里浮现出形如“const int”这样的字符串就 OK 了。而看到 declaration-specifiers，我们应联想到“static const int”。

```
struct-declaration:
    specifier-qualifier-list struct-declarator-list ;
specifier-qualifier-list:
    type-specifier specifier-qualifier-listopt
    type-qualifier specifier-qualifier-listopt
struct-declarator-list:
    struct-declarator
    struct-declarator-list , struct-declarator
```

至此，我们介绍了 C 标准文法中的 3 类声明符和对应的 3 类声明，它们是：

(1) 用于描述全局或局部变量的 declarator，对应的声明是 declaration。非终结符 declarator 中包含了标志符。

(2) 用于描述类型相关信息的 abstract-declarator，对应的声明是 parameter-declaration。引入 abstract-declarator 的目的是删去 declarator 中的标志符。

(3) 用于描述结构体中某成员的 struct-declarator，对应的声明是 struct-declaration。引入 struct-declarator 的目的是为了支持结构体位域成员。

而为了支持全局变量和局部变量的初始化，引入了 init-declarator 的概念，与之相关的产生式如下所示。看到 init-declarator，我们脑子里闪现的应是形如“`a = 3`”这样的式子。

```
init-declarator:
    declarator
    declarator = initializer
```

我们知道，在 C 语言中可以有如下所示的初始化，结构体对象和数组的初始化其实是比较复杂的，C 编译器在语义检查时也是要花费一番功夫的。

```
int a = 3;
int b[10][20] = { {1,2},{3,4},{5,6,7}};
```

我们需要引入一个称为“初值 (initializer)”的概念来刻画形如“`3`”和“`{ {1,2},{3,4},{5,6,7}}}`”这样的用来进行变量初始化的值。与 initializer 相关的产生式如下所示：

```
initializer:
    assignment-expression
    { initializer-list }
    { initializer-list , }

initializer-list:
    initializer
    initializer-list, initializer
```

由上述产生式，我们可以知道，我们在 3.1 节介绍过的赋值表达式就可以作为一个 initializer 来使用，例如整数 `3`。而“初值列表 (initializer-list)”则是若干个由逗号隔开的 initializer 构成，例如“`3,4,5,6,7`”。如果给这个初值列表再加上一对大括号，则构成了

“`{3,4,5,6,7}`”，这又可以当作用于对一维数组进行初始化的 initializer。所以在上述产生式中，我们看到`{ initializer-list }`是 initializer 的一个候选式。而候选式`{ initializer-list , }`只是多加了一个逗号而已，纯属满足不同 C 程序员的不同审美观。

子曾经曰过，“必也正名乎”。在这一小节，我们对 C 语言的标准文法做了简要介绍，在熟悉这些概念后，再进行 `ucl\decl.c` 的代码阅读，就真的会如孔老夫子所说的“名正言顺”。谈到“名正言顺”，我们就顺便对之前还未提及的一个细节进行说明。在 UCC 的源代码中，我们看到形如 `ParseDeclarationSpecifiers()` 这样的函数名，但在 C 标准文法中，我们遇到的却是非终结符 `declaration-specifiers`，有时我们为了讨论方便，也直接用函数名中的 `DeclarationSpecifiers` 来代表正式文法中的非终结符 `declaration-specifiers`。而在 UCC 的源代码注释中，我们还会看到用形如`[declaration-specifiers]`这样的式子，表示 `declaration-specifiers` 是可选的，而在 C 标准文法中对应的却又是 `declaration-specifiersopt`。其原因很简单：在源代码编辑器中，我们没办法如 Word 那样标出下标 `opt` 来，于是就用一对方括号`[]`来表示“可选的”。

下一小节，就让我们名正言顺地去分析与声明相关的代码。阅读 C 标准文法的窍门在于，要想到这个非终结符是用来描述什么样的字符串，例如，看到 `declaration-specifiers`，我们能想到形如“`static const int`”就可以了，其他的事情就要相信我们自己大脑的联想能力了。表 3.1 列出了与声明有关的几个非终结符及其含义，其中 `struct-or-union-specifier` 在语法上的地位与 `int` 相当。

表 3.1 与声明有关的非终结符

非终结符	对应中文名	例子
declaration	声明	static const int a1=3, *a2, a3[5], a4(void);
declaration-specifiers	声明说明符	static const int
declarator	声明符	a1 或 * a2 或 a3[5] 或 a4(void)
init-declarator	可带初值的声明符	a5 = 3 或 a1 或* a2 或 a3[5] 或 a4(void)
abstract-declarator	抽象声明符	删去 declarator 中的 id
parameter-declaration	形式参数声明	int a 或 int
struct-or-union-specifier	结构或联合说明符	struct Data{const int a; const int b:20;}
struct-declaration	结构体成员声明	const int a; 或 const int b:20;
specifier-qualifer-list	说明符限定符列表	const int
struct-declarator	结构体成员声明符	b:20

### 3.3.3 声明说明符和声明符

在上一小节中，我们讨论了 C 标准文法中与“声明”有关的几个非终结符。在此基础上让我们看看 ucl\decl.c 中与“声明说明符”和“声明符”相关的代码。函数 ParseCommonHeader() 用于对“声明说明符”和“可带初值的声明符”进行语法分析。

```

1 static AstDeclaration ParseCommonHeader(void) {
2     AstDeclaration decl;
3     AstNode *tail;
4
5     CREATE_AST_NODE(decl, Declaration);
6     // declaration-specifiers
7     decl->specs = ParseDeclarationSpecifiers();
8     if (CurrentToken != TK_SEMICOLON) {
9         //a = 3, b , c=50;
10        // or
11        // f(int a, int b);
12        // [init-declarator-list]
13        decl->initDecls = (AstNode)ParseInitDeclarator();
14        tail = &decl->initDecls->next;
15        while (CurrentToken == TK_COMMA) {
16            NEXT_TOKEN;
17            *tail = (AstNode)ParseInitDeclarator();
18            tail = &(*tail)->next;
19        }
20    }
21    return decl;
22 }
```

图 3.42 ParseCommonHeader()

图 3.42 第 5 行创建了一个 kind 域为 NK\_Declaration 的语法树结点，第 7 行通过函数 ParseDeclarationSpecifiers() 为“声明说明符”创建了语法子树，而如果接下来遇到的是分号，则说明当前声明中不存在“可带初值的声明符”。如果“声明说明符”之后存在“可带初值的声明符”，则通过在第 13 行和第 17 行调用函数 ParseInitDeclarator()，为“可带初值的声明符”创建相应的语法子树，第 16 行用于跳过相邻声明符之间的逗号，其余的代码则把若干个 init-declarator 对应的语法子树链接到一起。图 3.43 给出了由函数 ParseCommonHeader() 构造的一棵语法树。

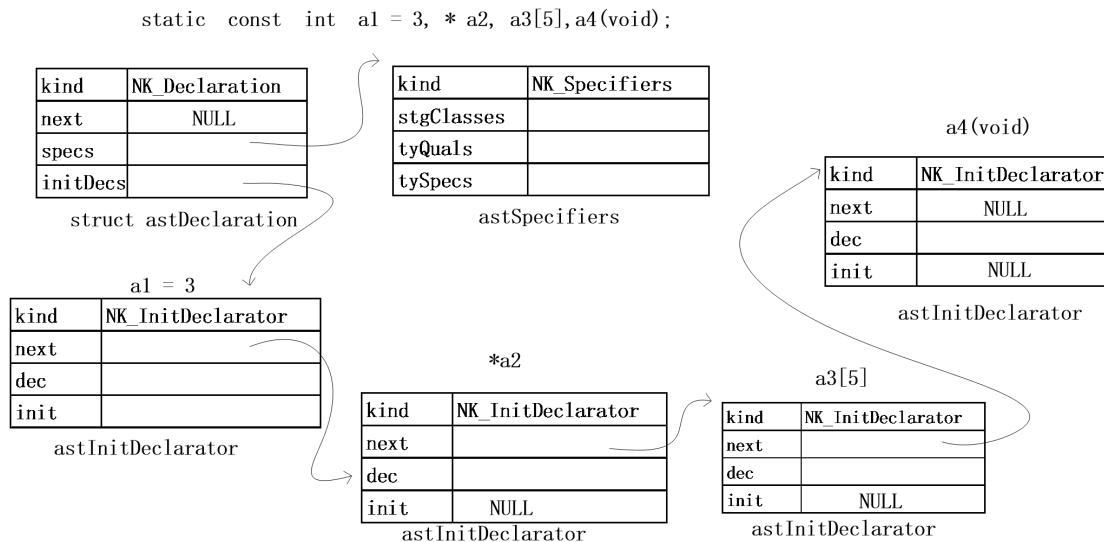


图 3.43 ParseCommonHeader() 构造的语法树

在图 3.43 中，我们有意选取与图 3.36 一样的例子，图 3.43 只给出了在函数 ParseCommonHeader() 中，我们所能看到的语法树轮廓，而语法树结点 astSpecifiers 和 astInitDeclarator 的相关细节，我们会在函数 ParseDeclarationSpecifiers() 和函数 ParseInitDeclarator() 中进行讨论。

让我们先讨论函数 ParseDeclarationSpecifiers()，图 3.44 给出了与“声明说明符”相关的产生式。第 7 至 12 行告诉我们，auto、register、static、extern 和 typedef 是存储类说明符；而由第 14 至 16 行可知，const 和 volatile 为类型限定符；而通过第 18 至 30 行可以发现，“int 等基本类型”、“结构或联合说明符”、枚举说明符和 TypedefName 等等都是“类型说明符”。第 2 和第 6 行规定了“声明说明符”是由若干个“存储类说明符（storage-class-specifier）”、“类型限定符（type-qualifier）”或者“类型说明符（type-specifier）”所构成。

```

1 /**
2  * declaration-specifiers:
3  *   storage-class-specifier [declaration-specifiers]
4  *   type-specifier [declaration-specifiers]
5  *   type-qualifier [declaration-specifiers]
6  *
7  * storage-class-specifier:
8  *   auto
9  *   register
10 *  static
11 *  extern
12 *  typedef
13 *
14 * type-qualifier:
15 *   const
16 *   volatile
17 *
18 * type-specifier:
19 *   void
20 *   char
21 *   short
22 *   int
23 *   long
  
```

```

24 *      float
25 *      double
26 *      signed
27 *      unsigned
28 *      struct-or-union-specifier
29 *      enum-specifier
30 *      typedef-name
31 */
32 static AstSpecifiers ParseDeclarationSpecifiers(void)

```

图 3.44 声明说明符的产生式

函数 ParseDeclarationSpecifiers() 的主要代码如图 3.45 所示，第 6 行创建了一个 kind 域为 NK\_Specifiers 的语法树结点，第 7 行的 specs->stgClasses 用于作为单向链表的链首，记录若干个“存储类说明符”结点构成的单链表；第 8 行的 specs->tyQuals 用于 volatile 或者 const 构成的单链表；第 9 行的 specs->tySpecs 所指向的单链表则记录了所有的类型说明符。

```

1 static AstSpecifiers ParseDeclarationSpecifiers(void) {
2     AstSpecifiers specs;
3     AstToken tok;
4     AstNode *scTail, *tqTail, *tsTail;
5     int seeTy = 0;
6     CREATE_AST_NODE(specs, Specifiers);
7     scTail = &specs->stgClasses;
8     tqTail = &specs->tyQuals;
9     tsTail = &specs->tySpecs;
10    nextSpecifier:
11        switch (CurrentToken) {
12            case TK_AUTO: ... case TK_TYPEDEF: ... break;
13            case TK_CONST: case TK_VOLATILE: ... break;
14            case TK_VOID: ... case TK_UNSIGNED: ... break;
15            case TK_ID:
16                // A typedefed name is also treated as type-specifier
17                if (!seeTy && IsTypedefName(TokenValue.p)) {
18                    ...
19                    break;
20                }
21                return specs;
22            case TK_STRUCT:
23            case TK_UNION:
24                // struct-or-union-specifier is also considered as type-specifier
25                *tsTail = (AstNode)ParseStructOrUnionSpecifier();
26                tsTail = &(*tsTail)->next;
27                seeTy = 1;
28                break;
29            case TK_ENUM:
30                // enum-specifier is considered as type-specifier too.
31                *tsTail = (AstNode)ParseEnumSpecifier();
32                tsTail = &(*tsTail)->next;
33                seeTy = 1;
34                break;
35            default:
36                return specs;
37        }
38        goto nextSpecifier;

```

```
39 }
```

图 3.45 ParseDeclarationSpecifiers()

对于 auto、const 和 int 等声明说明符的处理较为简单，我们在图 3.45 中省略了与之相关的代码，如第 12 至 14 行所示。当遇到一个标志符时，我们需要判断一下这个标志符是否为通过 typedef 定义来的类型名，第 15 至 21 行完成了这个判断。而第 22 至 28 行通过调用 ParseStructOrUnionSpecifier() 函数来实现对“结构或联合说明符”的分析；第 29 至 34 行则通过调用 ParseEnumSpecifier() 完成了对枚举说明符的分析。第 10 行的标号 next\_specifier 和第 38 行的 goto 语句构成了一个 while(1) 循环。若我们在第 17 行遇到一个不是类型名的标志符时，则说明 DeclarationSpecifiers 已经分析完了，我们会从第 21 行的 return 语句返回。而遇到其他不可能是“声明说明符”前缀的终结符时，则从第 36 行的 return 语句返回。图 3.46 给出了由 ParseDeclarationSpecifiers() 函数构造的一棵语法树。

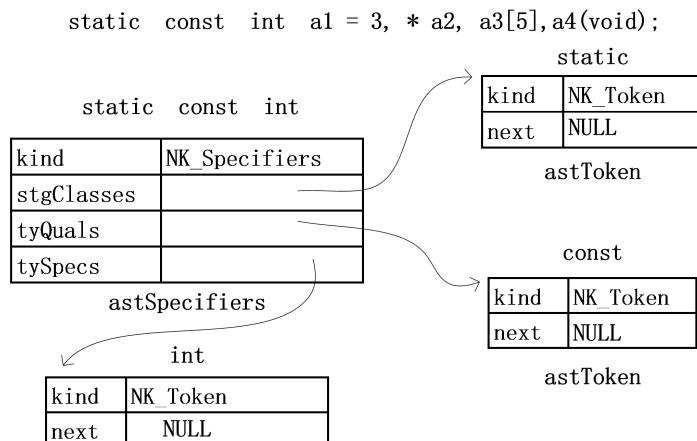


图 3.46 由 ParseDeclarationSpecifiers() 构造的语法树

接下来，让我们先看一下“结构或联合说明符”的产生式，如图 3.47 第 2 至 12 行所示。在语法上，C 语言的结构体和联合体的地位是一样的，两者的差别在于内存布局这样的语义上。第 6 行引入了非终结符 struct-or-union 来代表 struct 或者 union。

```

1 /**
2  * struct-or-union-specifier:
3  *     struct-or-union [identifier] { struct-declaration-list }
4  *     struct-or-union identifier
5 *
6  * struct-or-union:
7  *     struct
8  *     union
9 *
10 * struct-declaration-list:
11 *     struct-declaration
12 *     struct-declaration-list struct-declaration
13 */
14 static AstStructSpecifier ParseStructOrUnionSpecifier(void){
15     AstStructSpecifier stSpec;
16     AstNode *tail;
17
18     CREATE_AST_NODE(stSpec, StructSpecifier);
19     if (CurrentToken == TK_UNION){
20         stSpec->kind = NK_UnionSpecifier;
21     }
22     NEXT_TOKEN;

```

```

23 switch (CurrentToken) {
24 case TK_ID:
25     /*****
26     struct ABC;
27     or
28     union ABC;
29     *****/
30     stSpec->id = TokenValue.p;
31     NEXT_TOKEN;
32     if (CurrentToken == TK_LBRACE) {
33         goto lbrace;
34     }
35     return stSpec;
36
37 case TK_LBRACE:
38 lbrace:
39     /*****
40     struct ABC{ // we are here now
41     }
42 //略

```

图 3.47 ParseStructOrUnionSpecifier() 产生式

图 3.47 第 2 至 4 行的产生式告诉我们，如下所示的 3 个声明说明符都是合法的。图 3.47 第 18 至 21 用于创建一个 kind 域为“NK\_StructSpecifier 或者 NK\_UnionSpecifier”的语法树结点，第 30 行在 stSpec->id 中记录结构体或者联合体的名字，如果紧随其后的是左大括号，则通过第 33 行的 goto 语句跳转到第 38 行。

```

struct {int a; int b; int c;}
struct Data {int a; int b; int c;}
struct Data2

```

而图 3.47 第 38 行之后的代码用于处理当前 token 是左大括号的情况，如图 3.48 所示。图 3.48 第 11 至 14 行用于处理形如“struct Data{}”这样的空结构体。GCC 和 Clang 允许空结构体，而微软的 cl.exe 遇到这种情况时，则会报错“error C2016: C 要求一个结构或联合至少有一个成员”。在语法分析时，支持空结构体是件很容易的事情，由于这样的空结构体对象所占内存大小为 0 字节，这会给后续的语义检查带来一些麻烦。图 3.48 第 15 至 24 行，则通过在第 17 行调用的函数 ParseStructDeclaration()，完成了对“结构体成员声明”的分析；第 19 行用于把结构体中各成员域对应的语法子树链接在一起，其链首被记录在第 15 行的 stSpec->stDecls。

```

1 case TK_LBRACE:
2 lbrace:
3     /*****
4     struct ABC{ // we are here
5     }
6     struct{ // or we are here now
7     }
8     *****/
9     NEXT_TOKEN;
10    stSpec->hasLbrace = 1;
11    if (CurrentToken == TK_RBRACE) {
12        NEXT_TOKEN;
13        return stSpec;
14    }
15    tail = &stSpec->stDecls;

```

```

16     while (CurrentTokenIn(FIRST_StructDeclaration)) {
17         *tail = (AstNode) ParseStructDeclaration();
18         if (*tail != NULL) {
19             tail = &(*tail)->next;
20         }
21         SkipTo(FF_StructDeclaration, "the start of struct declaration or ");
22     }
23     Expect(TK_RBRACE);
24     return stSpec;
25 default:
26     Error(&TokenCoord, "Expect identifier or { after struct/union");
27     return stSpec;
28 }
29 }

```

图 3.48 ParseStructOrUnionSpecifier()\_左大括号

相对结构体的分析函数 ParseStructOrUnionSpecifier()而言，枚举说明符的分析函数 ParseEnumSpecifier()会简单一些，我们就不在啰嗦。图 3.49 给出了通过 ParseStructOrUnionSpecifier() 函数构造的一棵语法树，其中 kind 域为 NK\_StructSpecifier 的结点对应的是整个 struct Data 结构体，而 kind 域为 NK\_StructDeclaration 的语法树结点对应的是结构体中的成员域。

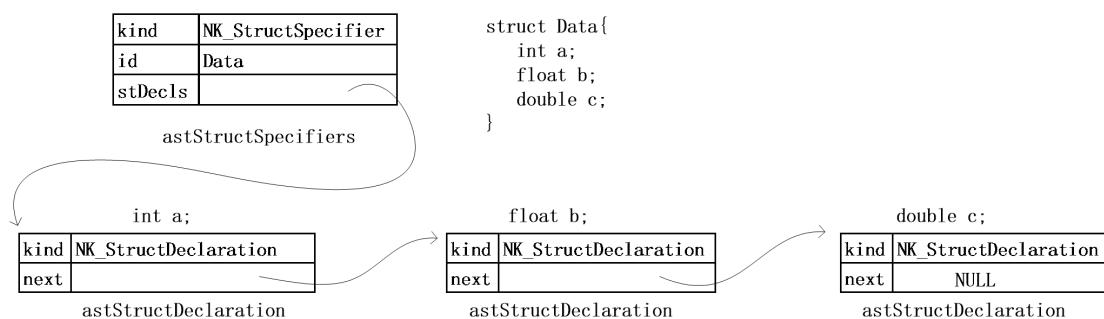


图 3.49 ParseStructOrUnionSpecifier() 构建的语法树

接下来，让我们看一下如何去分析“结构体成员声明”，相应的函数在 ParseStructDeclaration() 中，如图 3.50 所示。在前一小节中我们介绍过，在结构体成员声明中不允许出现形如 static 的存储类说明符，同时结构体成员还可以是位域成员，因此，我们没办法沿用之前的非终结符 declaration，而需要再定义一个非终结符 struct-declaration。当然，C 标准文法里的非终结符 struct-declaration 可能取名为 struct-field-declaration 或者 struct-member-declaration 会更恰当些。

```

1 /**
2  * struct-declaration:
3  *     specifier-qualifier-list struct-declarator-list ;
4  *
5  * specifier-qualifier-list:
6  *     type-specifier [specifier-qualifier-list]
7  *     type-qualifier [specifier-qualifier-list]
8  *
9  * struct-declarator-list:
10 *     struct-declarator
11 *     struct-declarator-list , struct-declarator
12 */
13 static AstStructDeclaration ParseStructDeclaration(void) {
14     AstStructDeclaration stDecl;

```

```

15 AstNode *tail;
16
17 CREATE_AST_NODE(stDecl, StructDeclaration);
18 // to support struct { ; }
19 if (CurrentToken == TK_SEMICOLON) {
20     NEXT_TOKEN;
21     return NULL;
22 }
23 /// declaration specifiers is a superset of specifier qualifier list,
24 /// for simplicity, just use ParseDeclarationSpecifiers() and check if
25 /// there is storage class
26 stDecl->specs = ParseDeclarationSpecifiers();
27 if (stDecl->specs->stgClasses != NULL) {
28     Error(&stDecl->coord, "Struct/union member should not have storage class");
29     stDecl->specs->stgClasses = NULL;
30 }
31 if (stDecl->specs->tyQuals == NULL && stDecl->specs->tySpecs == NULL) {
32     Error(&stDecl->coord, "Expect type specifier or qualifier");
33 }
34 // to support anonymous struct/union member in struct/union
35 if (CurrentToken == TK_SEMICOLON) {
36     NEXT_TOKEN;
37     return stDecl;
38 }
39 // struct-declarator-list
40 // struct-declarator,struct-declarator, ...., struct-declarator
41 stDecl->stDecls = (AstNode)ParseStructDeclarator();
42 tail = &stDecl->stDecls->next;
43 while (CurrentToken == TK_COMMA) {
44     NEXT_TOKEN;
45     *tail = (AstNode)ParseStructDeclarator();
46     tail = &(*tail)->next;
47 }
48 Expect(TK_SEMICOLON);
49 return stDecl;
50 }

```

图 3.50 ParseStructDeclaration()

在图 3.50 第 17 行，我们创建了一个 kind 域为 NK\_StructDeclaration 的语法树结点，第 26 行通过调用我们之前讨论过的函数 ParseDeclarationSpecifiers()，对结构体成员声明中的“声明说明符”进行分析。由于不允许出现形如 static 的存储类说明符，我们还需要在第 27 至 30 行做进一步检查；而“类型说明符”和“类型限定符”也不能同时为空，第 31 至 33 行对此进行了检查。第 34 至 38 行是为了允许形如在“struct Data{int a; float;}”中的“float;”这样的匿名成员域。在图 3.50 第 39 至 48 行，通过调用 ParseStructDeclarator() 完成了对结构体成员声明符的分析，第 46 行把若干个语法子树链接到一起，链首则记录在第 41 行的 stDecl->stDecls 中。对以下结构体 struct Data 中的成员声明“int a,b,c;”而言，图 3.51 给出了由函数 ParseStructDeclaration() 所构建的一棵语法树。

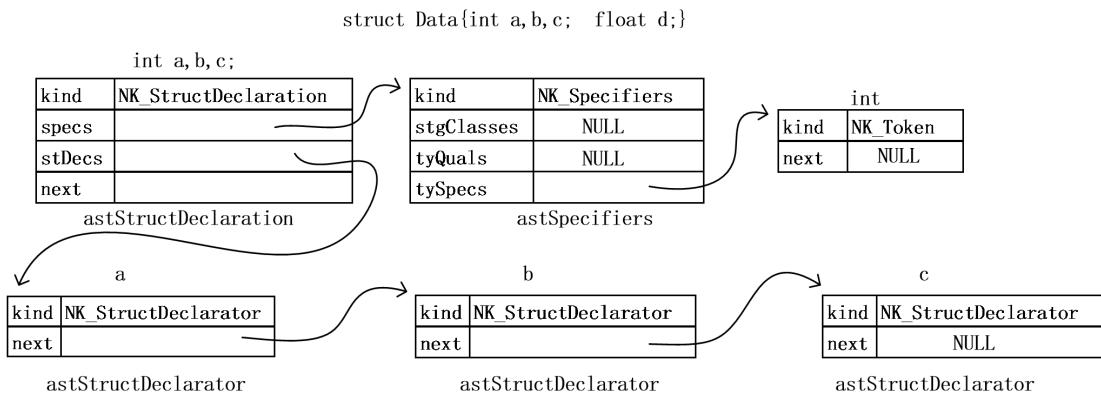


图 3.51 ParseStructDeclaration() 所构建的语法树

紧接着，让我们来看一下 ParseStructDeclarator() 所对应的代码，为了支持结构体中的位域成员，我们才引入了非结终符 struct-declarator，因此函数 ParseStructDeclarator() 的大部分功能是通过调用 ParseDeclarator() 来实现的，如图 3.52 所示。

```

1  ****
2  struct-declarator:
3      declarator
4      [declarator] : constant-expression
5  例子:
6  struct Data{
7      int :4; //合法
8      int a:12;
9      int b:4;
10     int c;
11 };
12 a:12      is struct-declarator
13 b:4      is struct-declarator
14 c      is struct-declarator
15 ****
16 static AstStructDeclarator ParseStructDeclarator(void){
17     AstStructDeclarator stDec;
18
19     CREATE_AST_NODE(stDec, StructDeclarator);
20     // [declarator]
21     if (CurrentToken != TK_COLON) {
22         stDec->dec = ParseDeclarator(DEC_CONCRETE);
23     }
24     // : constant-expression
25     if (CurrentToken == TK_COLON) {
26         NEXT_TOKEN;
27         stDec->expr = ParseConstantExpression();
28     }
29     return stDec;
30 }

```

图 3.52 ParseStructDeclarator()

图 3.52 第 2 至 14 行给出了与 struct-declarator 相关的产生式及对应的例子。第 19 行创建了一个 kind 域为 NK\_StructDeclarator 的语法树结点，如果当前终结符不是冒号，则在第 22 行调用函数 ParseDeclarator() 来对声明符进行分析。图 3.52 第 27 行调用 ParseConstantExpression() 来分析位域成员中的常量表达式。在 C 标准文法中，常量表达式被

定义为条件表达式，至于该条件表达式是否确实为常量，则需要在后续的语义分析时做进一步检查。

```
constant-expression:
    conditional-expression
```

至此，我们绕了一大圈，又回到了对“声明符”的分析。实际上在 ucl\decl.c 中，从头到尾，我们都是绕着“声明说明符”和“声明符”在转的。在前面的分析中，我们已经基本完成了对声明说明符的分析，现在，我们来看一下声明符的产生式，为了简化处理，UCC 中把 declarator 和 abstract-declarator 统一放在函数 ParseDeclarator() 中进行处理，改造后的文法如图 3.53 所示。

```
1  ****
2      abstract-declarator:
3          * [type-qualifier-list] abstract-declarator
4          postfix-abstract-declarator
5
6      postfix-abstract-declarator:
7          direct-abstract-declarator
8          postfix-abstract-declarator [ [constant-expression] ]
9          postfix-abstrace-declarator( [parameter-type-list] )
10
11     direct-abstract-declarator:
12         ( abstract-declarator )
13         ε
14
15     declarator:
16         * [type-qualifier-list] declarator
17         postfix-declarator
18
19     postfix-declarator:
20         direct-declarator
21         postfix-declarator [ [constant-expression] ]
22         postfix-declarator ( parameter-type-list )
23         postfix-declarator ( [identifier-list] )
24
25     direct-declarator:
26         ID
27         ( declarator )
28
29 @param kind
30     For function declaration:
31         DEC_CONCRETE|DEC_ABSTRACT
32     For function definition
33         DEC_CONCRETE
34     For type-name, sizeof(type-name), (type-name) (expr)
35         DEC_ABSTRACT
36 ****
37 static AstDeclarator ParseDeclarator(int kind)
```

图 3.53 声明符 Declarator 对应的产生式

图 3.53 第 2 至 13 行给出了“抽象声明符”的产生式，而第 15 至 27 行给出了与“声明符”有关的产生式。由于在函数声明中可以省略形参的名字，因此可以调用函数 ParseDeclarator(DEC\_CONCRETE|DEC\_ABSTRACT) 来分析其中的声明符，即传给函数 ParseDeclarator 的参数可以是 DEC\_CONCRETE|DEC\_ABSTRACT，如图 3.53 第 31 行所示；

而在 C 语言的函数定义中，形参名是不可以省略的，此时需要调用 ParseDeclarator(DEC\_CONCRETE)来进行语法分析，如图 3.53 第 33 行所示；而在需要类型名 TypeName 的场合，比如强制类型转换或者 sizeof 表达式中，我们不需要声明符的名称，使用的是抽象声明符，此时要调用 ParseDeclarator(DEC\_ABSTRACT)来进行分析，如图 3.53 第 35 行所示。在编译原理的相关教材中，一般会用一个特殊的符号  $\epsilon$  来表示空串。由图 3.53 第 13 行的  $\epsilon$  可知，对于抽象声明符而言，其声明符可以是匿名的。对“声明符”来说，我们可以发现在第 26 行确实有一个标志符 ID 来作为其名称。这也是“声明符”和“抽象声明符”最明显的区别。

“声明符”和“抽象声明符”的另一个区别体现在图 3.53 第 23 行。我们可以有以下旧式风格的函数定义，第 23 行的候选式是用来描述形如 `f(a,b,c)` 的字符串，其中的非终结符 identifier-list 描述了形如 “`a,b,c`” 这样的标志符列表。在“声明符”中允许出现这样的标志符列表，而由图 3.53 第 6 至 9 行，我们可以发现，在“抽象声明符”中不允许形如 “`a,b,c`” 这样的形参列表。

```
void f(a,b,c) int a,b;double c;{  
}
```

实际上，我们在第 1 章介绍 `ucc\examples\sc` 时就已介绍过“声明符”的语法分析。这里，我们就只讨论与第 1 章的例子有较大区别的地方。如图 3.54 所示，在第 1 章中我们只要处理第 2 至 4 行的 direct-declarator 就可以，而在 UCC 编译中，我们还需要处理第 6 至 8 行的 direct-abstract-declarator。

```
1  /**  
2   *  direct-declarator:  
3   *      ID  
4   *      ( declarator )  
5   *  
6   *  direct-abstract-declarator:  
7   *      ( abstract-declarator)  
8   *       $\epsilon$   
9   */  
10 static AstDeclarator ParseDirectDeclarator(int kind)  
11 {  
12     AstDeclarator dec;  
13     // ( declarator) or (abstract-declarator)  
14     if (CurrentToken == TK_LPAREN){  
15         NEXT_TOKEN;  
16         dec = ParseDeclarator(kind);  
17         Expect(TK_RPAREN);  
18         return dec;  
19     }  
20     // ID or NULL  
21     CREATE_AST_NODE(dec, NameDeclarator);  
22     if (CurrentToken == TK_ID){  
23         if (kind == DEC_ABSTRACT){  
24             Error(&TokenCoord, "Identifier is not permitted in the abstract declarator");  
25         }  
26         dec->id = TokenValue.p;  
27         NEXT_TOKEN;  
28     } else if (kind == DEC_CONCRETE){  
29         Error(&TokenCoord, "Expect identifier");  
30     }  
31 }
```

```

32     return dec;
33 }
```

图 3.54 ParseDirectDeclarator()

如果当前读头下的终结符是左括号，我们想尝试按照(declarator)或者(abstract-declarator)去做分析，图 3.54 第 15 行的 NEXT\_TOKEN 用于跳过左括号，而第 16 行递归地调用函数 ParseDeclarator(kind) 来分析 declarator 或者 abstract-declarator。

而如果当前读头下的符号是标志符 TK\_ID，但是我们要分析的却是抽象声明符，则说明这个标志符是多余的，图 3.54 第 23 至 25 行进行了错误处理；若我们要分析的确实是声明符，则需要在第 26 行记下这个标志符的名称。

如果当前读头下的符号即不是标志符也不是左括号，由图 3.54 第 2 至 4 行的产生式，我们很容易知道由左括号和标志符是 direct-declarator 的首符，若遇到其他终结符，则需要报错，如第 29 至 31 行所示。但是，如果我们是按第 6 至 8 行的产生式做推导，由于空串 $\epsilon$ 也是 direct-abstract-declarator 的候选式，这意味着我们可以不移动读头，即不需要调用 NEXT\_TOKEN。在由第 32 行返回的 kind 域为 NK\_NameDeclarator 的语法树结点中，对于抽象声明符，其 id 域为 NULL；对于声明符，我们已经在第 26 行记下了当前声明符的名称。

与第 1 章所举的 ucc\examples\sc 的另一个不同点在于函数 ParsePostfixDeclarator()，如图 3.55 所示。

```

1 static AstDeclarator ParsePostfixDeclarator(int kind){
2     AstDeclarator dec = ParseDirectDeclarator(kind);
3     while (1) {
4         if (CurrentToken == TK_LBRACKET) { // array , arr[3][4]
5             ...
6         } else if (CurrentToken == TK_LPAREN) { // function , f(...)
7             // notice: for abstract declarator, identifier list is not permitted,
8             // but we leave this to semantic check
9             AstFunctionDeclarator funcDec;
10            CREATE_AST_NODE(funcDec, FunctionDeclarator);
11            funcDec->dec = dec;
12            NEXT_TOKEN;
13            if (IsTypeName(CurrentToken)) { // f(int a, int b, int c);
14                funcDec->paramTyList = ParseParameterTypeList();
15            }
16            else{
17                funcDec->ids = CreateVector(4);
18                if (CurrentToken == TK_ID) { // f(a,b,c)
19                    INSERT_ITEM(funcDec->ids, TokenValue.p);
20                    NEXT_TOKEN;
21                    while (CurrentToken == TK_COMMA) {
22                        NEXT_TOKEN;
23                        if (CurrentToken == TK_ID) {
24                            INSERT_ITEM(funcDec->ids, TokenValue.p);
25                        }
26                        Expect(TK_ID);
27                    }
28                }
29            }
30            Expect(TK_RPAREN);
31            dec = (AstDeclarator) funcDec;
32        } else{
33            return dec;
34        }
35    }
36}
```

```

34      }
35  }
36 }
```

图 3.55 ParsePostfixDeclarator()

图 3.55 第 2 行调用了函数 ParseDirectDeclarator(), 而对于声明符中的后缀部分, 其中包含了数组类型和函数类型的相关信息。数组类型的处理和第 1 章类似, 我们在第 4 至 5 行中省略了相关代码; 与第 1 章有较大区别的是对函数参数列表的处理, 如第 6 至 34 行所示。由 C 的标准文法, 我们可知, 抽象声明符中不允许出现 f(a,b,c), 但是 UCC 为了语法处理上的方便和统一, 把这个检查推迟到语义检查时, 如第 7 至 8 行的注释所示。第 13 至 15 行通过调用函数 ParseParameterTypeList(), 处理了形如 f(int a,int b,int c)这样的新式风格的函数接口。而第 17 至 31 行则处理旧式风格的函数定义, 形如 f(a,b,c), 我们把标志符 a、b 和 c 记录在第 17 行所创建的向量中, 具体的插入操作由第 19 行和第 24 行的 INSERT\_ITEM 来完成。UCC 编译器在 ucl\vector.h 中提供的向量接口不算太复杂, 我们就不再啰嗦。而用于分析形参列表的函数 ParseParameterTypeList()如图 3.56 所示。

```

1 static AstParameterDeclaration ParseParameterDeclaration(void) {
2     AstParameterDeclaration paramDecl;
3     CREATE_AST_NODE(paramDecl, ParameterDeclaration);
4     paramDecl->specs = ParseDeclarationSpecifiers();
5     // f(int a, int b, int c); ---> CONCRETE
6     // or
7     // f(int, int , int)           ---> ABSTRACT
8     paramDecl->dec   = ParseDeclarator(DEC_ABSTRACT | DEC_CONCRETE);
9     return paramDecl;
10 }
11 /**
12 * parameter-type-list:
13 *     parameter-list
14 *     parameter-list , ...
15 * parameter-list:
16 *     parameter-declaration
17 *     parameter-list , parameter-declaration
18 * parameter-declaration:
19 *     declaration-specifiers declarator
20 *     declaration-specifiers abstract-declarator
21 */
22 AstParameterTypeList ParseParameterTypeList(void) {
23     AstParameterTypeList paramTyList;
24     AstNode *tail;
25     CREATE_AST_NODE(paramTyList, ParameterTypeList);
26     paramTyList->paramDecls = (AstNode)ParseParameterDeclaration();
27     tail = &paramTyList->paramDecls->next;
28     while (CurrentToken == TK_COMMA) {
29         NEXT_TOKEN;
30         if (CurrentToken == TK_ELLIPSIS) {
31             paramTyList->ellipsis = 1;
32             NEXT_TOKEN;
33             break;
34         }
35         *tail = (AstNode)ParseParameterDeclaration();
36         tail = &(*tail)->next;
37     }
}
```

```

38     return paramTyList;
39 }

```

图 3.56 ParseParameterTypeList()

由图 3.56 第 15 至 20 行的候选式可知，形参列表由若干个被逗号隔开的形参声明所构成，而形参声明又是由“declaration-specifiers declarator”或者“declaration-specifiers abstract-declarator”构成。由图 3.56 第 26 行和第 35 行，我们调用函数 ParseParameterDeclaration() 来对形参声明进行语法分析，该函数的代码在图 3.56 第 1 至 10 行。从中可以发现，最终我们调用的仍然是第 4 行的 ParseDeclarationSpecifiers() 和第 8 行的 ParseDeclarator() 这两个函数。所以，某种意义上，理解了 declaration-specifiers 和 declarator 这两个非终结符，我们就理解了 C 语言的声明，也初步理解了 C 语言的类型表达式。

在 declarator 的基础上，让我们看一下可带初值的 init-declarator 的相关代码，如图 3.57 所示。图 3.57 第 10 行调用了 ParseDeclarator(DEC\_CONCRETE) 函数来完成对声明符的分析，而如果之后的终结符是赋值号，则说明紧随其后的是初值，我们会在第 13 行调用函数 ParseInitializer() 来进行语法分析。

```

1  /**
2   *  init-declarator:
3   *      declarator
4   *      declarator = initializer
5   */
6  static AstInitDeclarator ParseInitDeclarator(void){
7      AstInitDeclarator initDec;
8
9      CREATE_AST_NODE(initDec, InitDeclarator);
10     initDec->dec = ParseDeclarator(DEC_CONCRETE);
11     if (CurrentToken == TK_ASSIGN) {
12         NEXT_TOKEN;
13         initDec->init = ParseInitializer();
14     }
15     return initDec;
16 }

```

图 3.57 ParseInitDeclarator()

与初值相关的产生式和分析函数如图 3.58 所示，其中第 15 行创建了一个 kind 域为 NK\_Initializer 的语法树结点，如果当前读头下的终结符不是左大括号，则我们会执行第 30 至 33 行的代码，第 31 行置 lbrace 为 0，表示没有左大括号，此时用于初始化的值是一个赋值表达式。而如果在第 17 行遇到的是左大括号，则执行第 18 至 29 行的代码，先在第 18 行置 lbrace 为 1，表示用于初始化的值是形如 {1,2,3} 这样的初值列表，第 20 至 28 行递归地调用函数 ParseInitializer() 来对初值列表中的各个初值进行语法分析。

```

1  /**
2   *  initializer:
3   *      assignment-expression
4   *      { initializer-list }
5   *      { initializer-list , }
6
7   *  initializer-list:
8   *      initializer
9   *      initializer-list, initializer
10  */
11 static AstInitializer ParseInitializer(void){
12     AstInitializer init;

```

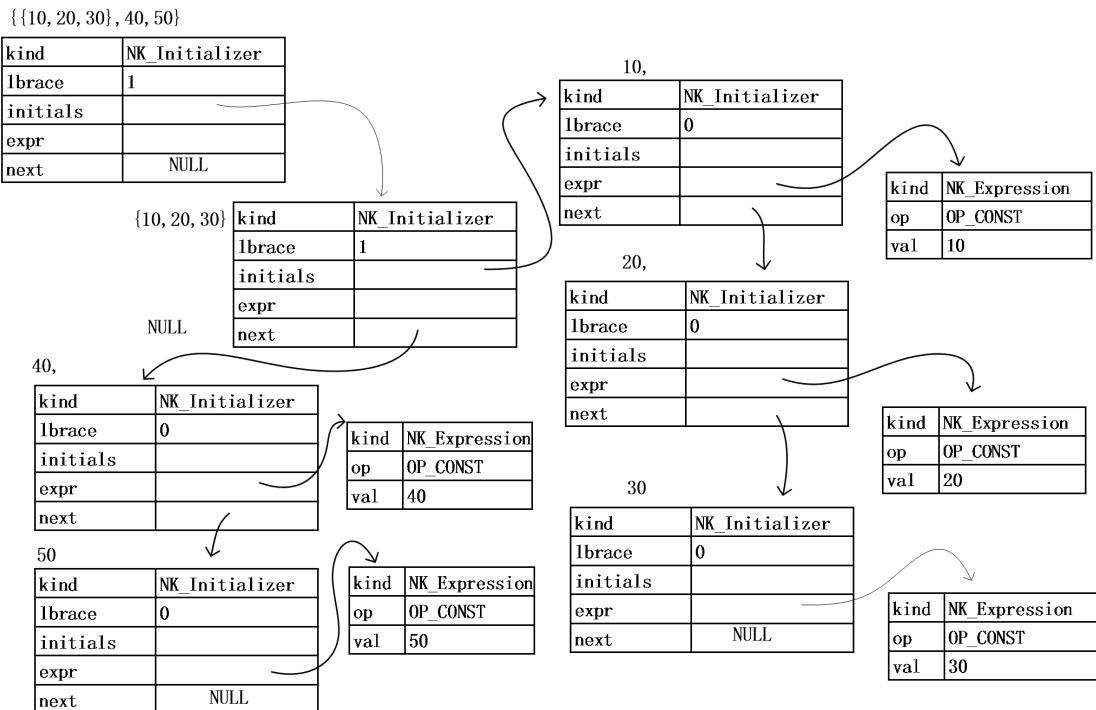
```

13 AstNode *tail;
14
15 CREATE_AST_NODE(init, Initializer);
16 // { 1,2,3},4,5,6
17 if (CurrentToken == TK_LBRACE) {
18     init->lbrace = 1;
19     NEXT_TOKEN;
20     init->initials = (AstNode)ParseInitializer();
21     tail = &init->initials->next;
22     while (CurrentToken == TK_COMMA) {
23         NEXT_TOKEN;
24         if (CurrentToken == TK_RBRACE)
25             break;
26         *tail = (AstNode)ParseInitializer();
27         tail = &(*tail)->next;
28     }
29     Expect(TK_RBRACE);
30 }else{
31     init->lbrace = 0;
32     init->expr = ParseAssignmentExpression();
33 }
34 return init;
35 }

```

图 3.58 ParseInitializer()

我们以 $\{\{10,20,30\},40,50\}$ 为例来看一下由图 3.58 所构造出来的语法树, 如图 3.59 所示, 其中 kind 域为 NK\_Initializer 的语法树结点记录了与初值的相关信息, initials 域和 expr 域在同一个联合体中, 为了区别起见, 我们把 initials 和 expr 分开画。当 lbrace 域为 1 时, 代表在该结点中记录的是初值列表的信息, 此时 initials 记录了各初值构成的链表的链首位置; 当 brace 域为 0 时, expr 指向一个表达式结点, 其中包含了一个初值。

图 3.59  $\{\{10, 20, 30\}, 40, 50\}$  对应的语法树

经过语法分析，最终我们得到的是形如前文图 3.34 的以 `astTranslationUnit` 对象为树根的语法树。如果输入的 C 源程序较复杂，则 C 编译器为之构造的语法树也是非常之庞大，很难把如此巨大的语法树完全画在纸上。相信结合图 3.34、图 3.36 和图 3.39 这样的简单例子，我们会在脑海中依稀建立起一棵语法树的轮廓。

按照 C 标准文法，编写大量的直接或间接递归的函数，进而为预处理后的 C 文件构建一棵语法树，这就是我们在语法分析这一章要完成的任务。最基本的分析方法，仍然是蕴含在第 1 章的例子 `ucc\examples\sc` 中。同时，我们也会发现，递归简直就是大自然的语言。虽然理论上，递归函数可以改成非递归函数，但在语法分析器的构造中，如果缺少了递归，那真的就会感觉无从下手。在下一章，我们将讨论 UCC 编译器的语义检查。

## 3.4 本章习题

1. 请仿照图 3.11 中的 `ParseBinaryExpression()` 函数，把第 1.3 节 `ucc\examples\sc` 中的函数 `MultiplicativeExpression()` 和 `AdditiveExpression()` 合并到一起。
2. 请仿照图 3.36，画出以下外部声明对应的语法树。  
`static const int a1 = 3, ** a2, a3[5][6], *a4(void);`
- 3 请仿照图 3.59，画出 `\{10,20,30\}, \{40,50\}, 60` 对应的语法树

# 第4章 语义检查

## 4.1 语义检查简介

在这一章中，我们需要在语法分析阶段建立的语法树的基础上，进行语义检查。UCC 编译器中与此相关的代码主要在 ucl\declchk.c、ucl\stmtchk.c 和 ucl\exprchk.c，分别用于对声明、语句和表达式进行语义检查。其中，最重要的工作就是建立 C 语言的类型系统，我们在第 2.4 节时就已简单介绍过 C 语言的类型系统。在阅读这部分代码时，一定需要结合语法树进行，看起来比较笨但却很管用的办法是，把第 3 章中我们构造出来的语法树画在纸质的笔记本上，对照着语法树，才不至于迷失方向。在如此枝繁叶茂的语法树上进行语义检查，如果没有对应的图纸做导航，很快就会“云深不知处”了。

按 UCC 编译器的正常执行流程，我们需要先为变量名和函数名建立起类型信息，这部分工作主要是在 declchk.c 中进行，之后再进行语句和表达式的语义检查。通过声明，C 程序员实际上建立了类型表达式，UCC 编译器会在语义检查时，为变量名和函数名等标志符建立起相应的类型结构。在第 2.4 节时，我们已初步介绍了在 UCC 编译器中，整数、函数、结构体、数组和指针等类型结构在 UCC 编译器内部是如何表示的。而 ucl\declchk.c 中的代码就是用于创建这些类型结构。有了这些类型信息后，我们就可以进一步检查一下表达式和语句在语义上是否正确，如图 4.1 所示。其中，a 是个结构体对象，而 b 是个整数，按 C 语言的语义，第 9 行的表达式 a+b 是非法的。对表达式的语义检查工作是在 ucl\exprchk.c 中进行的，第 16 行的错误提示告诉我们这个错误是在 exprchk.c 第 886 行发现的。由第 17 行的错误提示可知，第 10 行的 case 语句没有出现在 switch 语句，这个错误是在 stmtchk.c 的第 146 行发现的。类似的，在构建类型系统时，UCC 编译器也会进行语义检查，例如，第 15 行的错误提示告诉我们，第 7 行声明的 d 与之前声明的类型不相容，在第 6 行中 d 被声明为 double，而在第 7 行又被声明为 int，这个错误是在 declchk.c 的第 1587 行检测到的。

```

1 struct Data{
2     int num;
3 };
4 struct Data a;
5 int b;
6 double d;
7 int d;
8 int main(int argc,char * argv[]){
9     a+b;
10    case 8:
11        b++;
12        return 0;
13    }
14 //语义检查发现的几个错误
15 (declchk.c,1587):(hello.c,7):error:Incompatible with previous definition
16 (exprchk.c,886):(hello.c,9):error:Invalid operands to +
17 (stmtchk.c,146):(hello.c,10):error:A case label shall appear in a switch.

```

图 4.1 语义检查的例子

阅读 UCC 编译器语义检查相关代码的一个小诀窍就是，故意编写一些形如图 4.1 的有语义错误的 C 程序，上机运行后，观察 UCC 编译器是在哪里进行报错的，又是如何发现这

个错误的。相对而言，`declchk.c` 中的代码比较复杂。在进行代码分析时，我们还是暂时忽略“如何构建第 2.4 节各图所示的类型结构”这样的问题，即先不去分析 `declchk.c` 中的代码。我们假设形如第 2.4 节的类型结构已经构建完毕，在此基础上，先对表达式进行语义检查，即先讨论 `exprchk.c` 中的代码。这种分析策略的好处是，我们可以在对表达式进行语义检查时，进一步熟悉 C 语言的类型系统。而对语句的语义检查，则相对会简单许多，看看 `stmtchk.c` 中的代码行数就可略知一二。最后，我们再回到语义检查的重点戏“类型系统的建立”，即 `declchk.c`。简单而言，分析的顺序如下所示：

```
exprchk.c ---> stmtchk.c ---> declchk.c
```

这样的分析次序违背 UCC 编译器的执行顺序，但是应比较符合“先易后难，先感性后理性”的理解和认知顺序。当然，这需要我们有“暂时囫囵吞枣和不求甚解”的洒脱，有了 2.4 节的基础，我们可以暂时搁置“如何建立类型系统”这样的拦路虎，而是先绕开它，直扑表达式和语句。如此迂回的目的不是为了避开它，而是为了更好地消灭它。回头翻看第 3 章的目录就会发现，其实我们在语法分析时，也是按照这样的次序来处理的。先使用类型系统来进行表达式的语义检查，然后再思考类型系统是如何创建的，这也比较符合大部分人的习惯。就如我们是先上机用 C 编译器来学 C 语言，到一定火候后，我们才会在这边讨论 C 编译器是如何构建的。

## 4.2 表达式的语义检查

### 4.2.1 表达式的语义检查简介

在这一节中，我们来分析一下 `exprchk.c` 中的代码。我们通过调用图 4.2 第 1 行的函数 `CheckExpression()`，对表达式所对应的语法树进行语义检查。虽然表达式对应的语法树结点的 `kind` 域皆为 `NK_Expression`，但它们的运算符 `op` 域还是不一样的。由此，我们可以为不同的运算符表达式构造不同的语义检查函数，然后把这些函数放置在如图 4.2 第 8 至 12 行所示的函数表 `ExprCheckers` 中，之后可用运算符 `op` 作为下标去取出相应的语义检查函数，如图 4.2 第 3 行所示。

```

1 //对表达式进行语义检查
2 AstExpression CheckExpression(AstExpression expr) {
3     return (* ExprCheckers[expr->op])(expr);
4 }
5 //OPINFO(OP_COMMA,           1,      ", ",        Comma,          NOP)
6 //OPINFO(OP_ASSIGN,          2,      "= ",       Assignment,    NOP)
7 //预处理前的代码
8 static AstExpression (* ExprCheckers[])(AstExpression) = {
9     #define OPINFO(op, prec, name, func, opcode) Check##func##Expression,
10    #include "opinfo.h"
11    #undef OPINFO
12 };
13 //经 ucc -E exprchk.c -o exprchk.i 预处理后的结果
14 static AstExpression (* ExprCheckers[])(AstExpression) = {
15     CheckCommaExpression,
16     CheckAssignmentExpression,
17     .....
18     CheckAssignmentExpression,
19     CheckConditionalExpression,
20     CheckErrorExpression,
```

```

21 CheckBinaryExpression,
22 .....
23 CheckBinaryExpression,
24
25 CheckUnaryExpression,
26 .....
27 CheckUnaryExpression,
28 CheckPostfixExpression,
29 .....
30 CheckPrimaryExpression,
31 CheckErrorExpression,
32 };

```

图 4.2 CheckExpression()

对于图 4.2 第 8 至 12 行的函数表 ExprCheckers，为了清楚起见，我们把经预处理后所得到的函数指针数组列在第 14 至 32 行。第 10 行通过包含头文件 opinfo.h，引入了与运算符有关的信息，这些信息如图 4.2 第 5 至 6 行所示。例如，第 5 行的 OP\_COMMA 对应的是运算符 op，接下来的 1 代表该运算符的优先级 prec 为 1，紧接着的","是用字符串表示的运算符名称 name，跟随其后的 Comma 是函数名 func，而最后的 NOP 对应的是运算符编码 opcode，会用于 UCC 中间代码中。头文件 opcode.h 列出了 UCC 编译器所有中间代码的运算符编码，我们会在讨论中间代码时再进行分析。第 15 行的 CheckCommaExpression 用于对逗号表达式进行检查，而第 21 行的 CheckBinaryExpression 则用于对二元运算符表达式进行检查。C 语言中的二元运算符较多，为了避免编写包含较多 case 的 switch 语句，UCC 编译器仍然采用函数表的方式来处理二元运算符，如图 4.3 所示。

```

1 static AstExpression (* BinaryOPCheckers[]) (AstExpression) = {
2     CheckLogicalOP,           // "||"
3     CheckLogicalOP,           // "&&"
4     CheckBitwiseOP,          // |
5     CheckBitwiseOP,          // ^
6     CheckBitwiseOP,          // &
7     CheckEqualityOP,         // ==
8     CheckEqualityOP,         // !=
9     CheckRelationalOP,        // >
10    CheckRelationalOP,       // <
11    CheckRelationalOP,       // >=
12    CheckRelationalOP,       // <=
13    CheckShiftOP,           // <<
14    CheckShiftOP,           // >>
15    CheckAddOP,              // +
16    CheckSubOP,              // -
17    CheckMultiplicativeOP,   // *
18    CheckMultiplicativeOP,   // /
19    CheckMultiplicativeOP   // %
20 };
21 //对二元表达式进行语义检查
22 static AstExpression CheckBinaryExpression(AstExpression expr) {
23     expr->kids[0] = Adjust(CheckExpression(expr->kids[0]), 1);
24     expr->kids[1] = Adjust(CheckExpression(expr->kids[1]), 1);
25     return (* BinaryOPCheckers[expr->op - OP_OR])(expr);
26 }

```

图 4.3 CheckBinaryExpression()

图 4.3 第 2 行的 CheckLogicalOP 用于对“逻辑或”和“逻辑与”进行语义检查，第 17

行的 CheckMultiplicativeOP 用于对乘、除和取余运算进行语义检查。形如“ $a+b$ ”的二元运算表达式都有一个运算符和两个操作数，第 23 行递归调用 CheckExpression(expr->kids[0]) 对左操作数进行语义检查，而第 24 行则递归调用 CheckExpression(expr->kids[1]) 对右操作数进行语义检查，第 25 行则通过函数表 BinaryOpCheckers 来对整个二元表达式“ $a+b$ ”进行语义检查。显然，我们是按照二叉树的后根遍历的次序来进行语义检查。

按照 C 的语义，当“数组类型和函数类型的操作数”参与运算时，有时需要进行类型的调整，我们在第 1.6 节介绍过数组名和函数名。如图 4.4 的函数 Adjust() 就用于此目的，图 4.4 第 9 至 11 行会把“函数类型”调整为“指向函数的指针类型”，由于转换之后的类型为指针类型，因此我们在第 11 行把 isfunc 置为 1，用来标记这个语法树结点原本对应的是一个函数，而不是指向函数的指针。

```

1  AstExpression Adjust(AstExpression expr, int rvalue) {
2      int qual = 0;
3      if (rvalue) {
4          qual = expr->ty->qual;
5          expr->ty = Unqual(expr->ty);
6          expr->lvalue = 0;
7      }
8
9      if (expr->ty->categ == FUNCTION) {
10         expr->ty = PointerTo(expr->ty);
11         expr->isfunc = 1;
12     } else if (expr->ty->categ == ARRAY) {
13         expr->ty = PointerTo(Qualify(qual, expr->ty->bty));
14         expr->lvalue = 0;
15         expr->isarray = 1;
16     }
17     return expr;
18 }
```

图 4.4 Adjust()

例如对于以下代码，通过对“函数定义”的分析和检查，我们会在符号表中记录函数名 f 的类型为“函数类型”，但在语句“FUNCPTR ptr = f;”中，则要对表达式 f 进行调整，使其类型为“指向函数的指针”。需要注意的是，符号表中 f 对应的类型始终为“函数类型”，只是表达式 f 对应的语法树结点的类型被调整为“指向函数的指针”。

```

void f(void) {
}
typedef void (*FUNCPTR)(void);
FUNCPTR ptr = f;
```

当图 4.4 第 1 行的参数 rvalue 为 1 时，表示我们可把参数 expr 当右值对待，此时，我们通过第 6 行把语法树结点 expr 的 lvalue 域置为 0，表示这个结点不是左值，而是充当右值来使用。例如对于以下代码来说，变量 a 和 b 都存放在“可读可写并且可供 C 程序员寻址”的内存单元中，它们都是左值。但在对“ $a+b$ ”进行语义检查时，由于参与加法运算的两个操作数在运算结束后都不会发生变化，即我们是按“只读”的方式来访问 a 和 b，把 a 和 b 所对应的语法树结点视为右值，在图 4.3 第 23 至 24 行调用 Adjust() 函数时，传给形参 rvalue 的实参都为 1。在 C 语言中，右值意味着“不可寻址或者只读可不写”，不可寻址的原因在于“这个值可能只是存放在一个临时变量或者一个寄存器中”。例如表达式“ $a+b$ ”的值。

```

int a,b,c;
c = a+b;
```

图 4.4 第 13 行则用于把“数组类型”调整为“指向数组元素的指针类型”。数组不可充

当左值，例如以下所示的语句“`arr = 0;`”是非法的，图 4.4 第 14 行会把 `arr` 对应语法树结点的 `lvalue` 域置为 0，表示该结点为右值。对数组进行类型调整后，我们在图 4.4 第 15 行把结点的 `isarray` 域置为 1，用来标记该语法树结点原本的类型是“数组类型”。对于以下代码，通过对声明“`int arr[4];`”的语义检查，我们会得到 `arr` 的类型为“`int [4]`”，并把这个类型信息记录到符号表中。在对表达式“`arr+1`”进行语义检查时，我们通过查找符号表可得 `arr` 的类型为“`int [4]`”。经过 `Adjust()` 函数的调整，表达式 `arr` 所对应的语法树结点的类型被调整为“`int *`”，但需要注意的是，符号表中的 `arr` 的类型信息仍然是“`int [4]`”。

```
int arr[4];
//对 arr+1 中的 arr, UCC 会调用 adjust()函数对其进行调整, 其类型为 int *
int * ptr = arr+1;
arr = 0;           //非法
//对 sizeof(arr)中的 arr, UCC 不调用 adjust()函数对其进行调整, 其类型仍为 int[4]
sizeof(arr);
```

我们在第 1.6 节时介绍过，C 语言的数组名和函数名会被特殊处理，图 4.4 中的 `Adjust()` 函数就是用于此目的。在调用 `Adjust(expr, rvalue)` 函数时，我们可根据上下文来决定参数 `rvalue` 的值，当参数 `rvalue` 为 1 时，可把结点 `expr` 当右值用，而 `rvalue` 为 0 时则不改变 `expr` 结点的左值或右值的属性。当然，如果 `Adjust()` 函数的参数 `expr` 是数组类型，则通过图 4.4 第 14 行，我们始终是将 `expr` 当右值来用，此时就不再乎参数 `rvalue` 的值是 1 还是 0。需要注意的是，并不是遇到数组名，UCC 编译器都会调用 `Adjust()` 函数进行类型调整，例如当遇到上述“`sizeof(arr);`”中的 `arr` 时，UCC 编译器并不会为之调用 `Adjust()` 函数。

结合在第 2.4 节给出的类型结构图，我们不难读懂图 4.4 第 13 行调用的 `PointerTo()` 和 `Qualify()` 函数。例如对于如下代码，`arr` 是个数组，`const` 说明整个数组的内容为只读，经过 `Adjust()` 类型调整后，我们期望“表达式 `arr[3]` 中的 `arr`”对应的语法树结点类型为 `const int *`，所以我们需要在图 4.4 第 4 行先记录下这个限定符 `const`，然后通过第 13 行的 `Qualify()` 和 `PointerTo()` 函数构建出类型 `const int *`。由此，表达式 `arr[3]` 对应的语法树结点的类型为 `const int`，当以下语句“`arr[3]=5;`”试图改变 `arr[3]` 时，会因无法通过类型系统的检查而报错“`error:The left operand cannot be modified.`”。

```
typedef int ARRAY[4];
const ARRAY arr = {11,12,13,14};
int main(int argc,char * argv[]){
    arr[3] = 5;
    return 0;
}
```

在这一小节中，我们介绍了对表达式进行语义检查所用到的 `ExprCheckers` 和 `BinaryOPCheckers` 这两张函数表，及对“函数和数组对应语法树结点”进行类型调整的函数 `Adjust()`。我们会在后面的章节中对 `ExprCheckers` 和 `BinaryOPCheckers` 这两张函数表做进一步分析。

## 4.2.2 数组索引的语义检查

在这一小节中，我们准备分析一下 `exprchk.c` 中的函数 `CheckPostfixExpression`，此函数用于对后缀表达式进行语义检查。后缀表达式所对应的语法树请参照第 3 章的图 3.21，这里不再重复。我们先从数组索引入手，图 4.5 给出了 `hello.c` 经 UCC 编译后生成的语法树 `hello.ast`，中间代码 `hello.ulil` 及最终生成的汇编代码 `hello.s` 的主要部分。

```
1 // hello.c
2 int arr[3][4];
3 typedef int (*ArrPtr)[4];
4 ArrPtr ptr = &arr[0];
```

```

5 int * ptr1 = &arr[0][0];
6 int ** ptr2 = &ptr1;
7 int main(int argc,char * argv[]){
8     ptr[1][2] = 1;
9     arr[1][2] = 2;
10    arr[0][0] = 3;
11    ptr2[0][0] = 5;
12    return 0;
13 }
14 // hello.ast
15 function main
16 {
17     (= ([] ([] ptr
18             16)
19             8)
20             1)
21     (= ([] ([] arr
22             16)
23             8)
24             2)
25     (= ([] ([] arr
26             0)
27             0)
28             3)
29     (= (* (+ (* (+ ptr2
30                     0))
31                     0))
32                     5)
33
34 }
35 //hello.uil
36 function main
37 t0 : ptr + 24;
38 *t0 = 1;
39 arr[24] = 2;
40 arr[0] = 3;
41 t4 :*ptr2;
42 *t4 = 5;
43 return 0;
44 ret
45
46 // hello.s
47 movl ptr, %eax
48 addl $24, %eax
49 movl $1, (%eax)
50 movl $2, arr+24
51 movl $3, arr+0
52 movl ptr2, %ecx
53 movl (%ecx), %edx
54 movl $5, (%edx)

```

图 4.5 数组索引的例子

在图 4.5 第 2 行中，我们定义一个二维数组 arr[3][4]，第 4 行的 ptr 是一个指向 int[4] 数组的指针，而第 5 行的 ptr1 是一个指向 int 的指针，第 6 行的 ptr2 是一个指向 int \* 的指针。第 8 至 11 行给出了对数组元素进行访问的一些后缀表达式。在 UCC 编译器内部构造的语法树形如第 3 章的图 3.21，当然在语义检查时，我们会对语法分析时构造的语法树进行一些修改，为了便于调试，UCC 编译器可以把语义检查后的语法树的主要部分打印出来，如图 4.5 第 14 至 34 行所示。例如，第 9 行的 “arr[1][2]=2;” 对应的抽象语法树在图 4.5 第 21 至 24 行。

图 4.5 第 21 至 22 行的([] arr 16) 表示一棵以[]运算符为根，左操作数为 arr，右操作数为 16 的语法树，我们不妨记这棵子树为 arr[16]。而 ([] ([] arr 16) 8) 则表示了一棵以[]运算符为根，左操作数为子树 arr[16]，右操作数为 8 的抽象语法树，我们不妨把 16 和 8 加一起，记这棵子树为 arr[24]。而 (= ([] ([] arr 16) 8) 2) 则表示一棵以赋值运算符=为根，左操作数为 arr[24]，右操作数为 2 的抽象语法树。为了可读性，UCC 编译器在打印(= ([] ([] arr 16) 8) 2) 时，增加了一些回车换行和缩进，如下所示。

```

arr[1][2] = 2;
(= ([] ([] arr
        16)
        8)
        2)

```

稍微了解一下人工智能和 LISP 语言，我们就会发现图 4.5 第 17 至 32 行的抽象语法树与 LISP 语言非常相似，实质上 LISP 语言就是一种直接建立在抽象语法树上的语言。换言之，LISP 程序员编写的 LISP 代码，就是用字符串表示的抽象语法树。由图 4.5 第 2 行可知，arr 是二维数组，由 3 个 int[4] 数组构成的数组，若每个 int 占 4 个字节，则每个 int[4] 数组要占用 16 个字节，因此在语义检查时，我们实际上把 arr[1][2] 转换成 ([] ([] arr 1\*16) 2\*4)，即 ([]

([] arr 16) 8)。UCC 源代码 exprchk.c 中的函数 ScalePointerOffset(), 会根据数组 arr 的类型信息, 把数组索引值 1 和 2 进行倍乘放大, 进而得到 16 和 8。当然如果索引值是变量, 例如 arr[x][y], 则需要进行  $16*x+y*4$  的计算, 才能得到 arr[x][y]所在的内存单元。由于变量 x 和 y 的值没法在编译时确定, 所以对  $16*x+y*4$  的求值, 需要在运行时进行, 编译时只能生成相应的代码。在语义检查时, 我们就需要为  $16*x$  这样的倍乘运算构造相应的语法树结点。函数 ScalePointerOffset()的代码如图 4.6 所示。

```

1 static AstExpression ScalePointerOffset(AstExpression offset, int scale) {
2     AstExpression expr;
3     union value val;
4
5     CREATE_AST_NODE(expr, Expression);
6     expr->ty = offset->ty;
7     expr->op = OP_MUL;
8     expr->kids[0] = offset;
9     val.i[1] = 0;
10    val.i[0] = scale;
11    expr->kids[1] = Constant(offset->coord, offset->ty, val);
12    return FoldConstant(expr);
13 }
```

图 4.6 ScalePointerOffset()

对于 arr[x][y]而言, 我们会进行  $16*x$  的计算, 需要调用 ScalePointerOffset(x, 16), 图 4.6 第 5 至 11 行构建了一个根结点为 OP\_MUL 的语法树, 第 8 行把 x 作为左操作数, 而第 10 行则把倍乘系数 16 作为右操作数。当面对 arr[1][2]时, 图 4.6 第 1 行的参数 offset 为编译时的常量 1, 此时我们完全可以在编译时进行  $1*16$  的乘法计算, 没有必要把这个计算推迟到运行时, 第 12 行的函数 FoldConstant()完成了这个被称为“常量折叠”的操作。函数 FoldConstant()的代码在 fold.c 中, 虽然较长但很好理解, 我们就不再啰嗦。

由图 4.5 我们可以发现, 虽然在语法上第 10 行和第 11 行的两条赋值语句几乎一模一样, 但经语义检查后, 生成的语法树却完全不同。其原因在于两者的寻址模式完全不一样, 对于 arr[0][0]而言, 在汇编指令中, arr 就是数组的首地址, 而 arr+0 就是元素 arr[0][0]的地址, 图 4.4 第 51 行的 “movl \$3, arr+0” 实现了 “arr[0][0]=3” 的赋值功能。但对 ptr2[0][0]而言, ptr2 是个 int \*\*类型的指针, ptr2[0][0]的含义实际上是 \*\*ptr2, 这需要进行两次的间接寻址, 才能得到所要的内存单元。因此, 在语义检查时, 我们需要把表达式 ptr2[0][0]的语法树改为  $(* (+ (* (+ ptr2 0)) 0))$ , 其中的子树  $(* (+ (* (+ ptr2 0)) 0))$  完成了第一次间接寻址, 此处的 \* 表示的是 dereference 而不是乘法运算, 这个英文单词经常被译为“解引用”或者“提领”。把 dereference 译为“提领”应是源于侯捷先生的相关 C++译作中, 初看“提领”之词, 似乎很唐突, 但仔细品味, 确实是再形象不过。就如去逛商场时寄存物品, 把私人物品存入寄存柜后, 我们会得到一个小纸条, 上面或者是箱号或者是一个条形码, 之后我们凭着这个小纸条去“提取或领取”我们寄存的物品, 而这个小纸条就相当于 C 语言中的地址, 所以“提领”之译实在是妙不可言。

```

arr[0][0] = 3;
ptr2[0][0] = 5;
//对应语法树
(= ([] ([] arr 0) 0) 3)
(= (* (+ (* (+ ptr2 0)) 0)) 5)
```

图 4.5 第 8 至 11 行中各语句对应的寻址模型如图 4.7 所示, 指针变量 ptr2、ptr1 和 ptr 都有相应的内存单元, 数组 arr 的起始地址为 0x804a070, 而 ptr1 和 ptr 内存单元中的内容都是 0x804a070, 这意味着这两个指针都指向数组 arr 的开始位置。而 arr[0][0]和 ptr2[0][0]对

应相同的内存单元,  $\text{ptr}[1][2]$  和  $\text{arr}[1][2]$  对应相同的内存单元。要通过指针  $\text{ptr}$  定位到  $\text{ptr}[1][2]$ , 我们需要先计算出  $\text{ptr}+1*16+2*4$ , 即  $\text{ptr}+24$ , 然后再由这个地址进行“提领”运算即可, 相应的中间代码如图 4.5 第 37 至 38 行所示, 而汇编代码如第 47 至 49 行所示。而要由  $\text{ptr2}$  定位到  $\text{ptr2}[0][0]$ , 则需要经过  $\text{ptr2}+0*4$  的运算, 此处  $0*4$  中的 4 来源于  $\text{sizeof}(\text{int}^*)$  为 4, 编译器进行常量折叠后, 即为  $\text{ptr2}$ , 然后进行  $*(\text{ptr2})$  的提领操作, 即得到  $\text{ptr1}$  中的内容  $0x804a070$ , 再进行  $0x804a070+0*4$  的运算, 其中  $0*4$  中的 4 是由于  $\text{sizeof}(\text{int})$  为 4, 之后再根据常量折叠后的  $0x804a070$  进行提领操作。与之对应的中间代码如图 4.5 第 41 至 42 行所示, 而汇编代码如图第 52 至 54 行所示。而  $\text{arr}[0][0]$  和  $\text{arr}[1][2]$  的定位则不需要在运行时进行“提领”操作, 要访问的内存单元的首地址在编译时就可以计算出来, 即  $\text{arr}$  和  $\text{arr}+1*16+8$  (也就是  $\text{arr}+24$ ), 对应的中间代码如图第 39 和 40 行所示, 汇编代码如图 50 和 51 行所示。

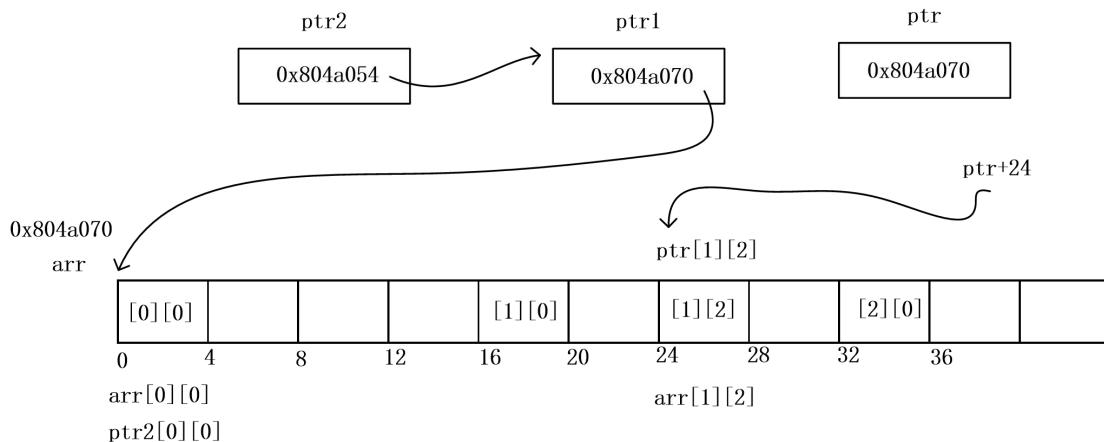


图 4.7 数组寻址

我们会在后续章节讨论中间代码和汇编代码的生成, 这里把相应的中间代码和汇编代码放在一起讨论的目的是为了进一步熟悉这些概念。对  $\text{arr}[0][0]$  和  $\text{ptr2}[0][0]$  而言, 在语法分析后两者的语法树是相似的, 但由上述分析可见, 在语义检查后, 两者语法树有较大差别。引起这种区别的源头在于  $\text{arr}$  和  $\text{ptr2}$  具有不一样的类型, 在对  $\text{arr}$  和  $\text{ptr2}$  的声明进行语义检查后, 填入符号表的  $\text{arr}$  类型为  $\text{int}[3][4]$ , 而  $\text{ptr2}$  的类型是  $\text{int}^{**}$ 。

经过这些准备工作后, 现在让我们来看看对数组索引进行语义检查的相关代码, 如图 4.8 所示。图 4.8 第 4 至 31 行完成了对数组索引运算符  $\text{OP\_INDEX}$  的语义检查, 而第 32 至 33 行调用  $\text{CheckFunctionCall(expr)}$  对“函数调用运算符  $\text{OP\_CALL}$ ”进行语义检查。成员选择运算符  $\text{OP\_MEMBER}$  和  $\text{OP\_PTR\_MEMBER}$  则通过第 36 行的  $\text{CheckMemberAccess(expr)}$  来处理。而“后加加”与“后减减”则通过在第 39 行调用  $\text{TransformIncrement(expr)}$  来检查。我们会在稍后对这几个函数进行分析。

```

1 static AstExpression CheckPostfixExpression(AstExpression expr)
2 {
3     switch (expr->op) {
4         case OP_INDEX:
5             expr->kids[0] = Adjust(CheckExpression(expr->kids[0]), 1);
6             expr->kids[1] = Adjust(CheckExpression(expr->kids[1]), 1);
7             if (IsIntegType(expr->kids[0]->ty)) {
8                 SWAP_KIDS(expr);
9             }
10            if (IsObjectPtr(expr->kids[0]->ty) && IsIntegType(expr->kids[1]->ty)) {
11                expr->ty = expr->kids[0]->ty->bty;
12                expr->lvalue = 1;
13                expr->kids[1] = DoIntegerPromotion(expr->kids[1]);
14                expr->kids[1] = ScalePointerOffset(expr->kids[1], expr->ty->size);
}

```

```

15         if(!expr->kids[0]->isarray && expr->ty->categ != ARRAY ){
16             AstExpression deref, addExpr;
17             CREATE_AST_NODE(deref, Expression);
18             CREATE_AST_NODE(addExpr, Expression);
19             deref->op = OP_DEREF;
20             deref->ty = expr->kids[0]->ty->bty;
21             deref->kids[0] = addExpr ;
22             addExpr->op = OP_ADD;
23             addExpr->ty = expr->kids[0]->ty;
24             addExpr->kids[0] = expr->kids[0];
25             addExpr->kids[1] = expr->kids[1];
26             deref->lvalue = 1;
27             return deref;
28         }
29         return expr;
30     }
31     REPORT_OP_ERROR;
32     case OP_CALL:
33         return CheckFunctionCall(expr);
34     case OP_MEMBER: // dt.a
35     case OP_PTR_MEMBER: // ptr->a
36         return CheckMemberAccess(expr);
37     case OP_POSTINC: // a++
38     case OP_POSTDEC: // a--
39         return TransformIncrement(expr);
40     default:
41         assert(0);
42     }
43     REPORT_OP_ERROR;
44 }
```

图 4.8 CheckPostfixExpression()

对于图 4.5 第 8 行的 `arr[1][2]` 而言，我们先来分析 `arr[1]`，如果把 `[]` 看成是运算符，则该后缀运算符实际上也有两个操作数，一个是 `arr`，另一个是 `1`。我们需要在图 4.8 第 5 至 6 行先对 `[]` 运算符的左右子树进行语义检查，这通过递归地调用 `CheckExpression()` 来完成。此处，左操作数 `arr` 和右操作数 `1` 实际上已是一个基本表达式，真正被调用的函数是 `CheckPrimaryExpression()` 函数，我们在稍后会对这个函数进行分析。在此函数中，会查符号表，得到 `arr` 的类型为 `int [3][4]`。此时，按照 C 的语义，`arr` 对应的语法树结点的类型需要被调整为指向数组元素的指针类型，对数组类型 `int [3][4]` 而言，其数组元素的类型为 `int[4]`，所以 `arr` 对应语法树结点的类型被调整为 `int (*)[4]`。图 4.8 第 5 至 6 行调用的函数 `Adjust()` 完成了这个类型调整的工作，我们已经在上一小节中讨论过函数 `Adjust()`。令大多数 C 程序员大跌眼镜的是“`1[arr][2] = 1;`”竟然也是合法的 C 语句，其等效于“`arr[1][2] = 1;`”。实际开发中，恐怕只有孔乙己穿越到现代才能写出如此奇葩的 C 语句来。不过 C 编译器还是要支持之，图 4.8 第 7 至 9 行用于把 `1[arr]` 转换成 `arr[1]` 来处理。坦白讲，实在看不出 C 编译器支持如此语法有何意义，权当满足“数学上的可交换性”和“孔乙己对茴香豆的茴有 4 种写法”的特殊癖好吧。当 `[]` 的左操作数是指向对象的指针类型（即除了函数指针外的其他指针类型），而右操作数是整数类型，则第 10 行的条件成立，我们就可进入第 11 行进行处理；否则跳到第 31 行进行错误处理，此时 `[]` 的左右操作数的类型不正确。这里，我们能看到类型系统再次派上用场。由于 `arr` 对应结点的类型已经被调整为 `int (*)[4]`，同时该结点的 `isarray` 域也会在 `Adjust()` 函数中被置为 1，代表 `arr` 对应的语法树结点的原有类型为数组类型。图

4.8 第 11 行把[]对应的语法树结点的类型置为 int[4]，而第 13 行则把右操作数 1 进行整型提升，即把 short 或 char 类型提升为 int 类型，第 14 行则调用我们前面讨论过的 ScalePointerOffset() 函数来实现  $1 * 16$  的运算，其中 `sizeof(int[4])` 正好就是 16。由于 arr 对应的结点的 isArray 域被置为 1，且[]对应的语法树结点的类型 int[4]也是数组类型，所以第 15 行的条件不成立，此时我们直接从第 29 行返回。这样，我们就完成了对 arr[1] 的语义检查，我们会返回如下语法树，其根结点的类型为 int[4]，即 arr[1] 的类型为 int[4]。

```
([] arr 16)
```

按照后根遍历的次序，接下来我们会再对 arr[1][2] 进行检查，此时左子树为 arr[1]，而右子树为 2，类似的，我们会得到以下语法树，其根结点的类型为 int。图 4.8 第 12 行会标记该语法树根结点为左值，此处即 arr[1][2] 可充当左值，故 arr[1][2]=1 是合法的。当然，按照图 4.8 的代码，如果要对 “arr[1] = 6;” 进行语义检查，我们会在第 12 行把 arr[1] 对应结点的 lvalue 域也误设为 1，但因为 arr[1] 的类型为数组类型 int[4]，在检查赋值运算符=时，即在后续要分析的函数 CheckAssignmentExpression() 中，我们会通过 Adjust() 函数，再次把 arr[1] 结点对应的 lvalue 置为 0，这表示 arr[1] 不可充当左值，所以还是会报错“error:The left operand cannot be modified”。

```
([] ([] arr 16) 8)
```

同理，对于 ptr[1][2] 和 arr[0][0] 来说，图 4.8 第 15 行的条件都不会成立，我们分别得到以下语法树。

```
([] ([] ptr 16) 8)
([] ([] arr 0) 0)
```

但是，对 ptr2[0][0] 而言，图 4.8 第 15 行的条件就会成立。我们先来看 ptr2[0]，查符号表可得 ptr2 原来的类型就是 int \*\*，而 ptr2[0] 对应的语法树结点的类型就应是 int \*，即 ptr2 不是数组类型且 ptr2[0] 也不是数组类型，所以第 15 行的条件会成立。图 4.8 第 16 至 27 行的代码则用于构造语法树(\* (+ ptr2 0))。按后根遍历的次序，完成对 ptr2[0] 的检查后，我们会再对 ptr2[0][0] 进行检查，此时 ptr2[0] 为左子树，而 0 为右子树，同理可得到如下语法树。

```
(* (+ (* (+ ptr2 0)) 0))
```

ptr2[0][0] 仍然可访问到数组元素 arr[0][0]，但如果想通过 ptr2[1][2] 来访问 arr[1][2] 则就是大错特错。因为 ptr2[1][2] 在语义检查后，对应的语法树为：

```
(* (+ (* (+ ptr2 4)) 8))
```

按图 4.7 所示，ptr2 的内容为 0x804a54，而 ptr2+4 的结果为 0x804a58，之后根据地址 0x804a58 做“提领”操作，天知道该单元中存放的是什么内容。

### 4.2.3 基本表达式的语义检查

在这一小节，我们来分析一下基本表达式的语义检查，primary-expression 的产生式如下所示，可以发现，“加了一对小括号的表达式”在语法上的地位跟标志符、常量和字符串是一样的。

```
primary-expression:
  ID
  constant
  string-literal
  ( expression )
```

例如，对于表达式  $(a+b)+c$  而言， $(a+b)$  和  $c$  都是基本表达式，但经语法分析后，我们为  $(a+b)+c$  生成的抽象语法树如下所示：

```
(+ (+ a b) c)
```

对  $(+ a b)$  这棵进行加法运算的语法树而言，运算符 + 是二元运算符，其语义检查是在函数 CheckBinaryExpression() 中完成的，而语法子树  $c$  则是在函数 CheckPrimaryExpression() 中

完成。换言之，在对基本表达式进行语义检查时，我们只需要考虑标志符、字符串和常量，而不需要考虑(expression)。我们先来看一下 UCC 编译器是如何处理字符串的，如图 4.9 所示。

```

1 // hello.c
2 #include <stdio.h>
3 char buf[] = "123456";
4 char * ptr = "654321";
5 int main(int argc,char * argv[]){
6     char buf2[] = "abcdef";
7     char * ptr2 = "fedcba";
8     printf("Hello World.\n");
9     return 0;
10 }
11
12
13 // hello.s
14 .data
15 .str0: .string "654321"
16 .str1: .string "fedcba"
17 .str2:.string"Hello World.\012"
18 .str3: .string "abcdef"
19 .globl buf
20 buf: .string "123456"
21 .align 4
22 .globl ptr
23 ptr: .long .str0
24
25 .text
26 .globl main
27 main:
28 .....
29 leal -8(%ebp), %edi
30 leal .str3, %esi
31 movl $7, %ecx
32 rep movsb
33 leal .str1, %eax
34 movl %eax, -12(%ebp)
35 leal .str2, %ecx
36 pushl %ecx
37 call printf
38 addl $4, %esp

```

图 4.9 字符串

图 4.9 第 1 至 11 行的 hello.c 中共有 5 个字符串，由第 15 至 20 行可见，除了用于初始化全局数组 buf[] 的“123456”外，UCC 编译器为其他的几个字符串取名为.str0、.str1、.str2 和.str3。而对字符串“123456”而言，我们就用全局字符数组 buf 的名称来为之命名，如图 4.9 第 20 行所示。但对第 6 行的局部数组 buf2[] 来说，其对应的存储空间在栈中，是在运行时动态分配，因此 UCC 编译器为第 6 行的字符串“abcdef”取名为.str3。而 buf2[] 数组的初始化则需要在运行时由第 29 至 32 行的多条汇编指令来完成，与之对比的全局数组 buf[] 的初始化在编译时就已经完成，如图第 19 至 20 行所示。第 3 行的字符串“123456”和第 6 行的字符串“abcdef”，分别用于初始化形如 buf[] 和 buf2[] 这样的字符数组，在 C 语言中，这是初始化数组的一种特殊用法，对此类字符串的语义检查，会在 declchk.c 的 CheckInitializerInternal() 函数中，即在对数组初始化进行检查时一并处理，而不会调用 exprchk.c 中的 CheckPrimaryExpression() 函数。上述 hello.c 中的其他字符串，都不是用于初始化字符数组，UCC 编译器会调用 CheckPrimaryExpression() 函数为这些字符串命名。对 C 程序员而言，这些字符串就相当于“匿名的”字符数组，如图 4.9 第 15 至 17 行所示。

图 4.10 给出了用于对基本表达式进行语义检查的函数 CheckPrimaryExpression()。对于形如 123 这样的常量，在语法分析时我们就已经在其语法树结点中记录了类型信息，在语义检查时，不需要做其他工作，由图 4.10 第 4 行直接返回即可。而对于字符串，UCC 编译器会为之取一个形如“.str1”这样的名称，并且将字符串加入到一个链表中，以便在代码生成时，能生成如图 4.9 第 15 至 17 行那样的汇编代码。这个工作主要由图 4.10 第 8 行调用的函数 AddString() 来完成。被 UCC 编译器命名后的字符串，就相当于具有了标识符的语法地位，所以第 7 行将字符串对应语法树结点的 op 域改为 OP\_ID。在 C 语言中，我们还可以读取字符串的地址，例如 printf("%p\n", &"abc")，这相当于字符串“abc”具有 C 程序员可见的内存地址，因此我们可将字符串当作左值来对待，在第 9 行将相应结点的 lvalue 域置为 1。

```

1 static AstExpression CheckPrimaryExpression(AstExpression expr) {
2     Symbol p;
3     if (expr->op == OP_CONST){ // 123

```

```

4     return expr;
5 }
6 if (expr->op == OP_STR) { // "abc"
7     expr->op = OP_ID;
8     expr->val.p = AddString(expr->ty, expr->val.p, &expr->coord);
9     expr->lvalue = 1;
10    return expr;
11 }
12 p = LookupID(expr->val.p);
13 if (p == NULL) {
14     Error(&expr->coord, "Undeclared identifier: %s", expr->val.p);
15     p = AddVariable(expr->val.p, T(INT),
16                     Level == 0 ? 0 : TK_AUTO, &expr->coord);
17     expr->ty = T(INT);
18     expr->lvalue = 1;
19 } else if (p->kind == SK_TypedefName) {
20     // typedef int INT32;      INT32 = 3;
21     Error(&expr->coord, "Typedef name cannot be used as variable");
22     expr->ty = T(INT);
23 } else if (p->kind == SK_EnumConstant) {
24     // enum Color{ RED, GREEN, BLUE};   int c = RED;
25     expr->op = OP_CONST;
26     expr->val = p->val;
27     expr->ty = T(INT);
28 } else{ // ID
29     expr->ty = p->ty;
30     expr->val.p = p;
31     expr->inreg = p->sclass == TK_REGISTER;
32     expr->lvalue = expr->ty->categ != FUNCTION;
33 }
34 return expr;
35 }

```

图 4.10 CheckPrimaryExpression()

图 4.10 第 12 至 34 行用于处理标志符，这需要由第 12 行的函数 `LookupID()` 查一下符号表，看看标志符是否已经声明过，如果是未声明就使用则在第 14 行报错，紧接着在第 15 行通过函数 `AddVariable()` 往符号表里添加一个 `int` 型的符号，这是“将错就错”的策略，以便后续的语义检查能继续进行下去。如果把通过 `typedef` 定义得来的类型名当作变量来使用，如第 20 行注释中的“`INT32 = 3;`”所示，则在第 21 行进行报错。对于形如第 24 行注释中的枚举常量 `RED`，我们可以把它当作整数常量来处理，如第 25 至 27 行所示。对于其他的标识符，我们把从符号表中查找得来的类型信息复制到语法树结点上，如第 29 至 31 行所示。函数名不可充当左值，第 32 行设置函数名结点的 `lvalue` 域为 0。而对数组名相应的语法树结点来说，若在语义检查时调用了 `Adjust()` 函数来做类型调整，则会在 `Adjust()` 函数中置其 `lvalue` 域为 0。例如以下的表达式 `arr+1`，在对二元运算符“`+`”进行语义检查时，我们会调用 `Adjust()` 来把数组名 `arr` 对应的语法树结点的类型调整为 `int *`，这样就可以进行 `arr+1` 的指针运算，而对于 `&arr` 而言，我们需要把 `arr` 当作左值来处理，这样可以取其地址，所以在对运算符 `&` 进行语义检查时，就不需要调用 `Adjust()` 函数。

```

int arr[4];
arr+1;
&arr;

```

在图 4.10 第 8 行调用的函数 `AddString()` 的代码如图 4.11 所示。我们要做的工作主要是

为这些字符串取一个名称，图 4.11 第 7 行的 FormatName() 函数完成了这个工作，这是一个 C 语言的变参函数，我们已在第 1.7 节介绍过 C 语言变参函数的实现原理。

```

1 Symbol AddString(Type ty, String str, Coord pcoord) {
2     Symbol p;
3
4     CALLOC(p);
5     p->pcoord = pcoord;
6     p->kind = SK_String;
7     p->name = FormatName("str%d", StringNum++);
8     p->ty = ty;
9     p->sclass = TK_STATIC;
10    p->val.p = str;
11
12    *StringTail = p;
13    StringTail = &p->next;
14    return p;
15 }
```

图 4.11 AddString()

图 4.11 第 4 行创建了一个 struct symbol 对象，第 5 至 10 行用于对这个符号对象进行初始化，第 6 行设置该符号为 SK\_String 类别，第 9 行的 TK\_STATIC 意味着这些“无名”的字符串其实被 UCC 编译器视为 static 的字符数组。在 UCC 编译器中，字符串并没有被添加到符号表里，而是用一个由若干个 struct symbol 对象构成的单向链表来记录，ucl\symbol.c 中的全局变量 Symbol Strings 记录了该链表的链首地址。图 4.11 第 12 行的 StringTail 始终指向这个链表的尾部，第 12 至 13 行完成了 struct symbol 对象的插入操作。

为了更清楚地对比语法分析后和语义检查后相关语法树结点的变化，我们给出了图 4.12，其中描绘了标志符结点 arr，常量结点 3 及索引结点 arr[3] 的对比情况。从中可以发现，op 域为 OP\_CONST 的常量结点在语义检查前后是没有发生变化的，而 op 域为 OP\_ID 的标识符结点却发生了一些变化。语义检查时，通过查符号表，我们为图中右侧的 arr 结点的 ty 域添上了类型信息。在符号表中 arr 的类型为 int [5]，但经过我们前面介绍的 Adjust() 函数的类型调整后，arr 结点的类型为 int \*，其 ty 域本应指向一个 struct type 对象。为了简单起见，我们直接在图中标上 int \*，其 val 域在语法分析后是指向字符串“arr”，但经语义检查后，我们让 val 指向标识符 arr 对应的 struct symbol 对象，在后续阶段进行中间代码和汇编代码生成时，我们需要用到 arr 对应 symbol 对象的内容。在图 4.12 左侧的 arr 结点中，我们没有画出其 isarray 和 ty 等域的内容，这些内容缺省值都为 0，因为我们在图 4.11 第 4 行从 UCC 的堆空间分配内存时，已把其中的内容清零。

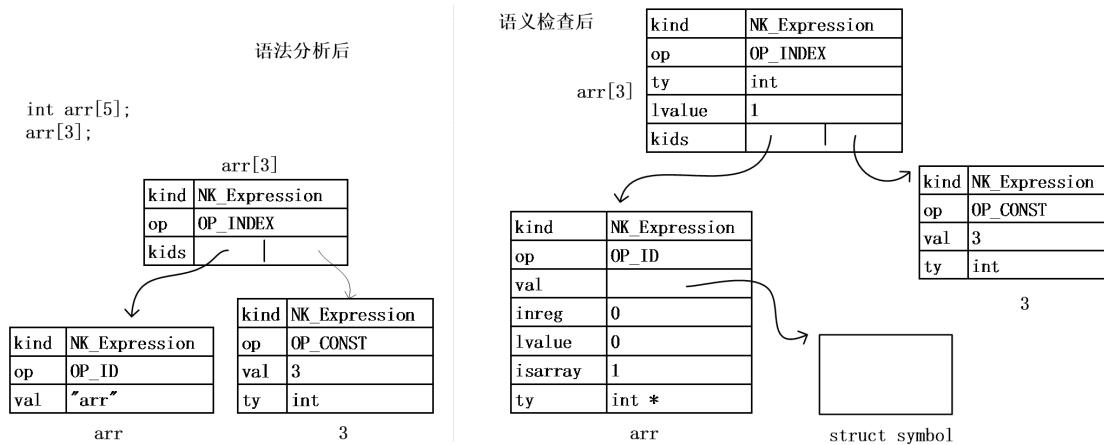


图 4.12 语法树的变化示意图

在调用函数 CheckExpression() 对 arr[3] 进行语义检查时，我们是会按后序遍历的次序来进行的，其原因是只有知道了 arr 对应结点的类型，我们才能知道 arr[3] 对应结点的类型。通过对声明 “int arr[5];” 的语义检查，我们建立了类型结构，这些类型信息保存在符号表中。在对表达式 arr[3] 进行语义检查时，我们需要从符号表中检索出 arr 对应的类型信息，然后让这些类型信息自下而上地在语法树上传播。结点 arr 的 op 域为 OP\_ID，因此，我们是用 CheckPrimaryExpression 函数来对之进行语义检查的。如图 4.10 第 32 行所示，在该函数中 arr 结点的 lvalue 会被置为 1，但是当 CheckPrimaryExpression() 返回时，我们会回到 CheckPostfixExpression() 函数中，图 4.8 第 5 行调用了 Adjust() 函数对左子树 arr 进行类型调整，从而使 arr 结点的 lvalue 为 0，其类型为 int \*，同时把 isArray 置 1，代表类型调整前 arr 结点为数组类型。由于 arr 结点的类型为 int \*，从而 arr[3] 结点的类型就为 int，且 arr[3] 结点的 lvalue 域为 1，表示该结点为左值。

#### 4.2.4 函数调用的语义检查

语法上，函数调用对应的表达式属于后缀表达式，如图 4.13 所示的 CheckFunctionCall() 用于对函数调用进行语义检查。而图 4.14 以 “f(30,40,50)” 为例，给出了语法分析和语义检查后的语法树。

```

1 static AstExpression CheckFunctionCall(AstExpression expr) {
2     AstExpression arg;
3     Type ty;
4     AstNode *tail;
5     int argNo, argFull;
6     if (expr->kids[0]->op == OP_ID
7         && LookupID(expr->kids[0]->val.p) == NULL) {
8         expr->kids[0]->ty = DefaultFunctionType;
9         expr->kids[0]->val.p =
10            AddFunction(expr->kids[0]->val.p, DefaultFunctionType,
11                         TK_EXTERN, &expr->coord);
12    }else{
13        expr->kids[0] = CheckExpression(expr->kids[0]);
14    }
15   expr->kids[0] = Adjust(expr->kids[0], 1);
16   ty = expr->kids[0]->ty;
17   if (! (IsPtrType(ty) && IsFunctionType(ty->bty))) {
18       Error(&expr->coord,
19              "The left operand must be function or function pointer");
20       ty = DefaultFunctionType;
21   }else{
22       ty = ty->bty;
23   }
24   tail = (AstNode *)&expr->kids[1];
25   arg = expr->kids[1];
26   argNo = 1;
27   argFull = 0;
28   while (arg != NULL && ! argFull){
29       *tail=(AstNode)CheckArgument(
30                  (FunctionType)ty,arg,argNo,&argFull);
31       tail = &(*tail)->next;
32       arg = (AstExpression)arg->next;
33       argNo++;

```

```

34 }
35 *tail = NULL;
36 while (arg != NULL) {
37     CheckExpression(arg);
38     arg = (AstExpression)arg->next;
39 }
40 argNo--;
41 if(argNo > LEN(((FunctionType)ty)->sig->params)){
42     if(((FunctionType)ty)->sig->hasProto){
43         if(!((FunctionType)ty)->sig->hasEllipsis){
44             Error(&expr->coord, "Too many arguments");
45         }
46     }else{
47         Warning(&expr->coord, "Too many arguments");
48     }
49 }else if (argNo < LEN(((FunctionType)ty)->sig->params)){
50     if(((FunctionType)ty)->sig->hasProto){
51         Error(&expr->coord, "Too few arguments");
52     }else{
53         Warning(&expr->coord, "Too few arguments");
54     }
55 }
56 expr->ty = ty->bty;
57 return expr;
58}

```

图 4.13 CheckFunctionCall()

对形如 `f(a,b,c)` 的函数调用进行语义检查时，我们需要先查找符号表，看看函数 `f` 是否已经声明过，图 4.13 第 6 至 7 行进行了这个判断。按 C 标准的规定，如果 `f` 未经声明就使用，则将函数 `f` 视为旧式风格的函数声明，相当于 `f` 被声明为 “`int f();`”。我们在第 2.4 节时已举过例子，旧式风格函数会带来令人抓狂的噩梦。这也应是 C++ 编译器禁止“函数未声明就使用”的原因。从这一点，我们也能再次体会，C++ 并非是 C 语言的超集，C++ 只是尽可能地去兼容已有的 C，对于实在看不下去的部分也采取了“摒弃”的策略。第 8 行的 `DefaultFunctionType` 则代表形如 `int f()` 的旧式风格函数所对应的类型，第 9 至 11 行把这个隐式声明的 `int f()`，通过函数 `AddFunction()` 添加到全局符号表中。当然，如果我们之前已经对函数 `f` 做了形如 `int f(int,int,int)` 的声明，此时就能在符号表中找到函数 `f` 的相关信息，或者当我们是通过形如 `(*ptr)(a,b,c)` 的方式来调用函数，此时表达式 `(*ptr)` 对应语法树结点的 `op` 域不为 `OP_ID`，则在第 13 行调用函数 `CheckExpression()` 来对表达式 `f` 或者 `(*ptr)` 进行语义检查。经过第 15 行的 `Adjust()` 函数的类型调整后，如果“`f` 或者 `(*ptr)` 对应结点”的类型不是“指向函数”的指针，那我们在第 18 至 20 行进行错误处理；否则，我们在第 22 行记下函数的类型信息。对于图 4.14 中的 `f` 对应的结点来说，其类型为“指向函数 `int (int,int,int)`”的指针，即 `int (*)(int,int,int)`，因此，我们在图 4.13 第 22 行记下的就是形如 `int(int,int,int)` 的类型信息，第 56 行则记下了函数返回值的类型，即表达式 `f(30,40,50)` 的类型为 `int`。关于函数类型的数据结构，请参考第 2 章的图 2.22。

图 4.13 第 24 至 39 行用于对函数调用中的各个实参进行检查，主要的工作由第 29 行的 `CheckArgument()` 来处理，检查的内容包括实参数个数是否与形参数个数吻合，实参与形参在类型上是否匹配，这相当于要检查能否把实参赋值给形参。第 40 至 55 行在实参数个数和形参数个数不一致时，会给出警告或者错误提示。对于形如 `int f()` 的旧式风格的函数声明来说，形参列表并不是其函数接口的一部分，即第 42 行和第 50 行的 `hasProto` 为 0（不存在原型）。原

型的意思是“这是范本，我们得依样画葫芦，范本有几个形参，调用时就得有几个实参”。对于形如 `int f(int,int,int)` 的新式风格函数声明，我们就得照着原型来进行函数调用了，不然就要报错了。图 4.13 第 36 至 39 行的代码用于检查多余的实参，例如函数调用 `f(30,40,50,60,70)`，而在新式风格的声明 `int f(int,int,int)` 中，我们只声明了 3 个形参。第 29 行的 `CheckArgument()` 在处理新式风格函数的参数时，如果发现已经检查完 3 个实参了，就会置第 27 行的 `argFull` 为 1，此后不必再执行第 28 行的 `while` 循环。由于语义检查时，语法树结点会发生变化，甚至是重新构建，所以我们要在第 29 行在记录下 `CheckArgument()` 的返回值，而对于多余的实参，则只是在第 37 行调用 `CheckExpression()` 检查一下。

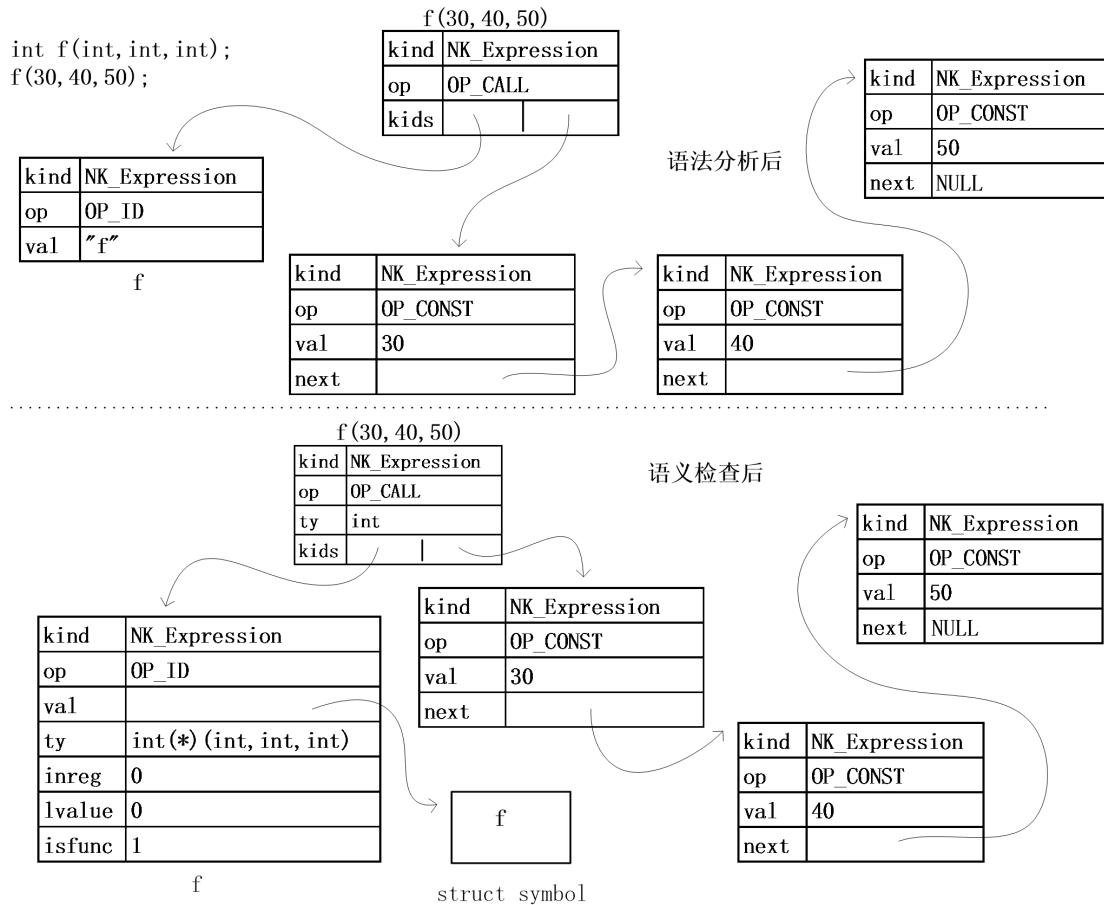


图 4.14 函数调用的语法树

接下来我们来分析一下图 4.13 第 29 行调用的函数 `CheckArgument()`，如图 4.15 所示。图 4.15 第 5 行用于获取新式风格的函数的形参个数，第 6 行调用函数 `CheckExpression()` 进行实参表达式的语义检查，如果新式风格的函数声明形如 `f(void)`，即不存在参数，则在第 8 行设置相应标志位为 1，表示对新式风格的函数实参已检查完毕，然后从第 9 行直接返回。对于形如 `f(int,int,int)` 的函数声明来说，`f` 不是变参函数，若当前要检查的实参是最后一个，则第 11 行的条件成立，此时在第 12 行置相应标志位为 1。而对于形如 `int f()` 的旧式风格的函数来说，形参列表并不是函数接口的一部分，我们需要对函数调用中的每个实参进行实参提升的操作，这由第 15 行的 `PromoteArgument()` 函数来完成。对于新式风格的函数，我们需要检查一下实参能否赋值给对应的有名形参，第 20 行的 `CanAssign()` 函数用来做这个判断。出于内存对齐的考虑，C 编译器通常会把小于 `int` 类型的实参（例如 `char` 或者 `short`）转换成 `int` 后入栈，第 23 行执行了这个由编译器隐式进行的转型操作。但是，对于新式风格函数中的 `float` 类型实参，并没有进行提升到 `double` 类型的动作，这与 `PromoteArgument()` 是有所区别的。第 28 行则是用于处理新式风格函数中的变参函数，用于对“无名参数”进行实参提升。

的操作。

```

1 static AstExpression CheckArgument(
2     FunctionType fty, AstExpression arg, int argNo, int *argFull)
3 {
4     Parameter param;
5     int parLen = LEN(fty->sig->params);
6     arg = Adjust(CheckExpression(arg), 1);
7     if (fty->sig->hasProto && parLen == 0) {
8         *argFull = 1;
9         return arg;
10    }
11    if (argNo == parLen && ! fty->sig->hasEllipsis) {
12        *argFull = 1;
13    }
14    if (! fty->sig->hasProto) // old style
15        arg = PromoteArgument(arg);
16        *argFull = 0;
17        return arg;
18    } else if (argNo <= parLen) // new style
19        param = GET_ITEM(fty->sig->params, argNo - 1);
20        if (! CanAssign(param->ty, arg))
21            goto err;
22        if (param->ty->categ < INT)
23            arg = Cast(T(INT), arg);
24        else
25            arg = Cast(param->ty, arg);
26        return arg;
27    } else{// variable arguments, ...
28        return PromoteArgument(arg);
29    }
30 err:
31 Error(&arg->coord, "Incompatible argument");
32 return arg;
33 }
```

图 4.15 CheckArgument()

图 4.15 第 20 行的 CanAssign() 函数是按照 C 标准文档 ansi.c.txt 来编写的，该文档规定了哪些情况下可以进行赋值操作，相关代码如图 4.16 所示。我们有意在第 9 至 12 行保留了一段来自 ansi.c.txt 的语义规则，第 13 行的 if 语句实现了这个判断，即如果赋值运算的左右操作数都是算术类型，则可以进行赋值。实参赋值给形参时可能需要的强制转型操作，我们已在图 4.15 第 22 至 25 行完成。

```

1 // see ansi.c.txt      3.3.16.1 Simple assignment
2 int CanAssign(Type lty, AstExpression expr) {
3     Type rty = expr->ty;
4     lty = Unqual(lty);
5     rty = Unqual(rty);
6     if (lty == rty) {
7         return 1;
8     }
9     /***** the left operand has qualified or unqualified arithmetic type and *****
10    the right has arithmetic type;
11    *****/
12 }
```

```

13  if (BothArithType(lty, rty)) {
14      return 1;
15  }
16  if (IsCompatiblePtr(lty, rty) && ((lty->bty->qual & rty->bty->qual) == rty->bty->qual)) {
17      return 1;
18  }
19  if ((NotFunctionPtr(lty) && IsVoidPtr(rty) || NotFunctionPtr(rty) && IsVoidPtr(lty)) &&
20      ((lty->bty->qual & rty->bty->qual) == rty->bty->qual)) {
21      return 1;
22  }
23  if (IsPtrType(lty) && IsNullConstant(expr)) {
24      return 1;
25  }
26  if (IsPtrType(lty) && IsPtrType(rty)) {
27      Warning(&expr->coord, "assignment from incompatible pointer type");
28      return 1;
29  }
30  if ((IsPtrType(lty) && IsIntegType(rty) || IsPtrType(rty) && IsIntegType(lty)) &&
31      (lty->size == rty->size)) {
32      Warning(&expr->coord, "conversion between pointer and integer without a cast");
33      return 1;
34  }
35  return 0;
36 }

```

图 4.16 CanAssign()

根据赋值运算的左操作数和右操作数的类型，我们在以下情况下可以进行赋值操作，这些情况需要按以下排列依次进行判断。

- (1) 两者类型一致，对应图 4.16 第 6 行。
- (2) 两者都是算术类型，对应图 4.16 第 13 行。
- (3) 两者是相容的指针类型，例如 T1 \* 和 T2 \*，其中 T1 和 T2 的类型相容，函数 IsCompatibleType()会对类型是否相容进行判断，我们会在后续章节对这个函数进行分析。图 4.16 第 16 行对此进行判断。
- (4) 两者都是指针类型，形如 T1 \* 和 T2 \*，T1 和 T2 中有一个是 void，但另一个不能是函数类型（即要求是结构体和 double 等描述数据的对象类型），对应图 4.16 第 19 行。
- (5) 左操作数的类型为指针类型，右操作数为常数 0，对应图 4.16 第 23 行。
- (6) 两者都是指针类型，对应图 4.16 第 26 行，此时在第 27 行给出一个警告。
- (7) 一个是指针类型，另一个是整数类型，且两者要占用同样大小的内存空间，对应图 4.16 第 30 行，此时在第 32 行给出一个警告。

从中，我们可以发现，不同类型的结构体对象是不能进行赋值的。接下来，我们来分析一下图 4.15 第 23 行用到的 Cast() 函数，其代码如图 4.17 所示。图 4.17 第 32 至 43 的函数 CastExpression() 用于构建转型运算的语法树结点，第 37 行创建一个语法树结点，第 39 行置其 op 域为 OP\_CAST，第 40 行记录转型后的结点类型，第 41 行则记录转型前的表达式。

当然，如果是对形如第 34 行的常量 3 进行转型，则可以在编译时进行简化，我们直接取 3.0f 即可，这个工作由 FoldCast() 函数完成，该函数在 fold.c 中，相对比较简单，我们就不再啰嗦。图 4.17 第 48 至 53 列出了 UCC 编译器内部的各种数据类型，第 2 行的 I4 表示要占用 4 个字节的有符号整数，在 32 位系统上对应 int 或者 long；U4 表示要占 4 个字节的无符号整数；F4 表示占 4 字节的浮点数，对应 float；F8 表示要占 8 字节的浮点数，V 表示 VOID，而 B 则代表 Block 对象，对应联合体、数组或结构体对象。第 55 行的 optypes[] 用于记录从

INT 到 I4 这样的映射关系，之后可调用第 44 行的函数 TypeCode() 来检索。第 2 行注释里的 I1 等类型编码是有先后次序的，按照这样的顺序，我们能比较快捷地进行类型判断，例如第 8 至 9 行的 if 语句就是用来判断“两个类型是否都是占用相同大小内存的整型”，short 和 unsigned short 就满足这个条件。

```

1  AstExpression Cast(Type ty, AstExpression expr) {
2      // enum {I1, U1, I2, U2, I4, U4, F4, F8, V, B};
3      int scode = TypeCode(expr->ty);
4      int dcode = TypeCode(ty);
5      if (dcode == V) {
6          return CastExpression(ty, expr);
7      }
8      if (scode < F4 && dcode < F4
9          && scode / 2 == dcode / 2) {
10         // see examples/cast/sign.c
11         int scateg = expr->ty->categ;
12         int dcateg = ty->categ;
13         if (scateg != dcateg && scateg >= INT
14             && scateg <= ULONGLONG){ // I4 与 U4 之间的转型
15             return CastExpression( ty,expr);
16         }
17         expr->ty = ty;           // I1 与 U1 之间, I2 与 U2 之间
18         return expr;
19     }
20     if (scode < I4){
21         expr = CastExpression(T(INT), expr);
22         scode = I4;
23     }
24     if (scode != dcode){
25         if (dcode < I4){
26             expr = CastExpression(T(INT), expr);
27         }
28         expr = CastExpression(ty, expr);
29     }
30     return expr;
31 }
32 static AstExpression CastExpression(Type ty, AstExpression expr) {
33     AstExpression cast;
34     // (float)3 -----> 3.0f
35     if (expr->op == OP_CONST && ty->categ != VOID)
36         return FoldCast(ty, expr);
37     CREATE_AST_NODE(cast, Expression);
38     cast->coord = expr->coord;
39     cast->op = OP_CAST;
40     cast->ty = ty;
41     cast->kids[0] = expr;
42     return cast;
43 }
44 int TypeCode(Type ty)
45 {
46     ****
47     see type.h
48     enum{

```

```

49      CHAR, UCHAR, SHORT, USHORT, INT, UINT,
50      LONG, ULONG, LONGLONG, ULLONG, ENUM,
51      FLOAT, DOUBLE, LONGDOUBLE, POINTER,
52      VOID, UNION, STRUCT, ARRAY, FUNCTION
53  };
54  ****
55 static int optypes[] = {I1, U1, I2, U2, I4,
56   U4, I4, U4, I4, U4,
57   /* float */ F4, F8, F8, U4, V, B, B, B};
58 assert(ty->categ != FUNCTION);
59 return optypes[ty->categ];
60 }

```

图 4.17 Cast()

当进行有 I1 与 U1 之间（或者 I2 与 U2 之间）的类型转换时，存放在内存单元的数据并没有发生任何变化。我们只要执行图 4.17 第 17 行的代码，在相应语法树结点上记录转型后的新类型即可。

在把 char 类型强制转换为 float 时，UCC 也是分两步走，第一步先把 char 提升为 int，之后再进行 int 到 float 的转型操作数。图 4.17 第 20 至 23 行完成了由 char 到 int 的提升，而第 28 行则进行了由 int 到 float 的转换。反之，把 float 类型强制转换成 char 类型时，UCC 也分两步走，第一步执行 float 到 int 的转换，第二步再进行 int 到 char 的转换，第 25 至 28 行完成了这两步的工作。而图 4.17 第 5 至 7 行可把表达式强制转换成 void。对于以下未被使用的函数参数 arg 来说，有些编译器会给出一个警告，避免这个警告的一个做法就是加上“(void) arg;”语句，如下所示。

```

void f(int arg){
    (void) arg;
    // ...
}

```

当进行 I4 与 U4 之间的类型转换时，图 4.17 第 15 行调用函数 CastExpression() 来显式地构造一个进行转型运算的语法树结点。该语法树结点会影响汇编代码生成时算术右移指令的选择。当进行有符号整数的算术右移时，在生成汇编指令时我们会选用 sar 指令，而对于无符号整数的算术右移则使用 shl 指令。下面我们举个例子来对图 4.17 做进一步解释，如图 4.18 所示。

```

1 // hello.c
2 #include <stdio.h>
3 short s = -1;
4 int b;
5 char c1;
6 float f;
7 int main(int argc,char * argv[]){
8     b = (int)((unsigned int)s) >> 1;
9     f = (float)c1;
10    c1 = (char)f;
11    return 0;
12 }
13
14 // hello.ast
15 function main{
16     (= b
17      (cast int
18       (>> (cast unsigned int

```

```

19             (cast int
20                     s))
21         1)))
22     (= f
23         (cast float
24             (cast int
25                 c1)))
26     (= c1
27         (cast char
28             (cast int
29                 f))))
30     (ret 0)
31 }

```

图 4.18 转型所对应的语法树

在 UCC 编译器中，参与算术运算的小于 int 的操作数 (I1、I2、I2 和 U2) 都会被提升到 int，例如图 4.18 第 8 行的 short 类型的变量 s。要注意的是即使是进行两个 char 类型变量的加法，例如 c1+c2，UCC 也是先把 c1 和 c2 提升为 int，然后做 32 位的加法运算。而进行强制类型转换时，例如图 4.18 第 9 行的(float) c1 和第 10 行的(char) f，我们也是以 int 类型作为中转，如图 4.18 第 22 至 29 行的语法树所示。理解了 Cast() 函数，那么再理解前文提及的用于实参提升函数 PromoteArgument()，就是件很容易的事情了。

```

static AstExpression PromoteArgument(AstExpression arg) {
    Type ty = Promote(arg->ty);
    return Cast(ty, arg);
}
Type Promote(Type ty) {
    return ty->categ < INT ? T(INT) : (ty->categ == FLOAT ? T(DOUBLE) : ty);
}

```

而对于前文提及的用于判断两个类型是否相容的函数 IsCompatibleType()，我们会在后续章节中进行讨论。要理解这个函数，我们需要对第 2.4 节中介绍的各种类型结构有感性的认识。对照着第 2.4 节给出的类型结构和第 3 章的语法树来阅读代码，才不至于在庞大的语法树上和复杂的类型结构里迷失方向。

前文中，有个比较微妙的函数是图 4.13 第 10 行的 AddFunction()。让我们结合图 4.19 中的例子来对此进行讨论。在函数 f 的函数体中，我们在图 4.19 第 3 行中对函数 h 进行了声明，这导致我们无法在第 5 行中再次声明 int 类型的变量 h。第 3 行对函数 h 的声明，与第 4 行对局部变量 a 的声明有点不同，第 4 行的 a 在函数 f 的函数体外是无法进行访问的。但第 3 行对 h 的声明，在不同编译器上却有着不同的结果。

```

1 #include <stdio.h>
2 void f(void){
3     int h(int,int);
4     int a;
5     //int h = 3;
6 }
7 int main(int argc,char * argv[]){
8     printf("%d \n", h(3.0,4.0));
9     return 0;
10 }
11 int h(int a,int b){
12     return a+b;
13 }

```

图 4.19 函数体中的函数声明

图 4.19 中的代码在 Clang3.0 和 VS2013 上得到的结果都是 7，这意味着第 8 行调用 `h(3.0,4.0)` 时，这些编译器是把 `h` 当作“`int h(int, int);`”来调用的，而不是旧式风格的“`int h();`”，真正执行的函数调用为“`h((int)3.0,(int)4.0);`”，压入栈的实参为 3 和 4。但 GCC 编译器在第 8 行调用函数 `h()` 时，认为 `h` 的接口为旧式风格的“`int h();`”，入栈的实参为 3.0 和 4.0，但在第 11 至 13 行的函数体 `h` 中，会把栈中 `double` 类型参数 3.0 分拆成两个 `int` 形参 `a` 和 `b` 来使用，由此得到结果 1074266112。在 C 语言的细枝末节，我们再次看到不同编译器呈现了完全不同的行为。在 UCC 编译器中，我们把第 3 行声明的函数 `h` 的符号填入局部符号表，同时也填入全局符号表，这样可产生与“Clang3.0 和 VS2013”相同的结果 7。

当然，如果我们删去第 3 行的声明“`int h(int,int);`”，则这些编译器都认为图 4.19 第 8 行调用的函数 `h` 是旧式风格的“`int h();`”，此时得到的结果都是 1074266112。

另外，对于以下两个函数声明来说，C 编译器会把它们视为彼此相容（compatible）的函数声明，函数 `IsCompatibleType()` 会用来检测两个类型是否相容。

```
int g(int (*)(), double (*)[3]);
int g(int (*)(char *), double (*)[]);
```

在这种情况下，C 编译器并不会报错，而是为这两个相容的声明类型，构造一个“相当于最大公因子的”类型，这个“最大公因子”在 C 标准文档中被称为合成类型（Composite Type）。对以上两个声明来说，最终合成的最大公因子如下所示。UCC 编译器 `type.c` 中的函数 `CompositeType(ty1,ty2)` 用于实现这个合成操作。我们会在后续章节对 `IsCompatibleType()` 和 `CompositeType()` 等函数做进一步分析。

```
int g(int (*)(char *), double (*)[3]);
```

由于有这样微妙的语义，在 UCC 编译器中，同一个函数名对应的符号可能要被加入到不同的符号表中。我们期望同一个函数名只对应一个 `symbol` 对象，当要通过 `CompositeType()` 函数来改变已有函数声明 `g` 的类型信息时，我们从符号表找到相应的符号对象就可以。正如第 2 章图 2.35 所述，为了能让同一个符号对象能出现在不同的符号表中，我们引入了结构体 `struct bucketLinker`。当一个符号对象需要被加入某个符号表时，我们就为之创建一个 `bucketLinker` 对象，因此，一个符号对象可对应多个 `bucketLinker` 对象。

```
typedef struct bucketLinker{
    struct bucketLinker * link;      // 用于构造哈希表里的哈希桶
    Symbol sym;                    // 指向符号对象
} * BucketLinker;
```

函数 `LookupID()` 对符号表的检索是从当前符号表开始，如果找不到，则再查找外层的符号表，直到全局符号表为止，其代码如图 4.20 第 20 至 22 所示。对符号表的查询操作实际上由第 1 行的 `DoLookupSymbol()` 函数来完成，第 3 行计算出哈希值，第 7 至 12 行的 `for` 循环根据这个哈希值在相应的哈希桶上进行查找。如果存在更外层的符号表，且我们想查找外层的符号表，则第 14 行的 `while` 条件会成立。

```
1 static Symbol DoLookupSymbol(Table tbl, char *name, int searchOuter) {
2     Symbol p;
3     unsigned h = (unsigned long)name & SYM_HASH_MASK;
4     BucketLinker linker;
5     do{
6         if (tbl->buckets != NULL) {
7             for(linker =(BucketLinker)tbl->buckets[h]; linker; linker = linker->link) {
8                 if (linker->sym->name == name) {
9                     linker->sym->level = tbl->level;
10                     return linker->sym;
11                 }
12             }
13         }
```

```

14 } while ((tbl = tbl->outer) != NULL && searchOuter);
15 return NULL;
16 }
17 static Symbol LookupSymbol(Table tbl, char *name){
18 return DoLookupSymbol(tbl, name, SEARCH_OUTER_TABLE);
19 }
20 Symbol LookupID(char *name){
21 return LookupSymbol(Identifiers, name);
22 }

```

图 4.20 LookupId() 函数

在 UCC 编译器的内部，用来存放符号的数据结构主要有两种，一种是哈希表，一种是单链表，uclsymbol.c 中定义的一些变量用于此目的，如图 4.21 所示。图 4.21 第 3 行的哈希表 GlobalTags 用于存放在全局作用域内声明的结构体、联合体和枚举的名称，这些名称在 C 标准文档中被称为“标签（Tag）”。而语句“goto Again”中的 Again 被称为标号（label），较易混淆的是 label 这个词也常被译为标签。而第 5 行的 GlobalIDs 用于存放全局的变量名和函数名，常量（例如 123）则存放于第 7 行的哈希表 Constants 中。由此，我们可以看到，即使是在同一作用域中声明的结构体名 struct Data 和变量名 int abc，也是存放在不同的哈希表中的。第 9 行的指针 Tags 指向了在当前作用域中用于结构体名的符号表，而第 11 行的指针 Identifiers 则指向了在当前作用域中存放变量名和函数名的符号表。为了后续阶段生成代码的方便，我们还会把函数名对应的符号链接在一起，其链首为第 16 行的 Functions，而全局变量和静态变量所在符号链的链首为第 18 行的指针 Globals。第 21 行的 Strings 和第 23 行的 FloatConstants 分别为字符串和浮点数的符号链的链首。而第 14 行的 FunctionTails 等指针则始终指向相应符号链的链尾，由此可方便地进行插入操作。

```

1 // 哈希表
2 // tags in global scope, tag means struct/union, enumeration name
3 static struct table GlobalTags;
4 // normal identifiers in global scope
5 static struct table GlobalIDs;
6 // all the constants
7 static struct table Constants;
8 // tags in current scope
9 static Table Tags;
10 // normal identifiers in current scope
11 static Table Identifiers;
12 // 单链表
13 // used to construct symbol list
14 static Symbol *FunctionTail, *GlobalTail, *StringTail, *FloatTail;
15 // all the function symbols
16 Symbol Functions;
17 // all the global variables and static variables
18 Symbol Globals;
19 // all the strings
20 // see EmitStrings(void)
21 Symbol Strings;
22 // all the floating constants
23 Symbol FloatConstants;

```

图 4.21 与符号有关的数据结构

对于通过 `typedef` 关键字建立的类型名，例如如下所示的“`typedef int Data;`”，UCC 编译器也会把 Data 存入由图 4.21 第 11 行的 Identifiers 所指向的当前符号表，而 `struct Data` 中的结构体名 Data 则存入由图 4.21 第 9 行 Tags 所指向的符号表。且在 C 语言中，使用以下结

构体名 Data 时，我们需要带上 struct 关键字，因此，在 C 语言中，我们可以无歧义地通过“Data a = 3;”把以下局部变量 a 声明为 Data 类型（即 int 型的别名），而非 struct Data 类型。而在 C++ 中，使用结构体或类名时，可以不需要 struct 或 class 关键字，这会引起在“Data a = 3;”中到底用哪个 Data 类型的歧义，因此 C++ 编译器会对以下代码报错。

```
void f(void) {
    struct Data{
        int a;
    };
    typedef int Data;
    Data a = 3;
}
```

函数 AddFunction() 的代码如图 4.22 所示。图 4.22 第 4 至 11 行创建了一个类别为 SK\_Function 的符号。如果当前所处作用域不是全局作用域，我们通过第 16 行的 AddSymbol() 函数，把符号 p 加入到当前符号表 Identifiers 中。UCC 编译器还会用一个单链表来记录所有函数名对应的符号，其链首为图 4.21 第 16 行的 Functions 变量，图 4.22 第 13 至 14 行用于在链尾插入符号 p。在 C 语言中，即使是在局部作用域中声明的函数，也被缺省地当作 extern 的函数声明，为了在图 4.19 上得到与“Clang 编译器和 VS2013”相同的结果，我们可把该函数名加入到全局符号表中，这由图 4.22 第 18 行调用的 AddSymbol() 函数来完成。

```
1 Symbol AddFunction(char *name,
2     Type ty, int sclass, Coord pcoord) {
3
4     FunctionSymbol p;
5     CALLOC(p);
6     p->kind = SK_Function;
7     p->name = name;
8     p->ty = ty;
9     p->sclass = sclass;
10    p->lastv = &p->params;
11    p->pcoord = pcoord;
12
13    *FunctionTail = (Symbol)p;
14    FunctionTail = &p->next;
15    if(Identifiers != &GlobalIDs) {
16        AddSymbol(Identifiers, (Symbol)p);
17    }
18    return AddSymbol(&GlobalIDs, (Symbol)p);
19 }
20
21 Symbol AddVariable(char *name,
22     Type ty, int sclass, Coord pcoord) {
23     VariableSymbol p;
24
25     CALLOC(p); p->kind = SK_Variable;
26     p->name = name; p->ty = ty;
27     p->sclass = sclass; p->pcoord = pcoord;
28     if (Level == 0 || sclass == TK_STATIC) {
29         *GlobalTail = (Symbol)p;
30         GlobalTail = &p->next;
31     } else if (sclass != TK_EXTERN) {
32         *FSYM->lastv = (Symbol)p;
33         FSYM->lastv = &p->next;
```

```

34 }
35 if(sclass == TK_EXTERN && Identifiers != &GlobalIDs) {
36     Symbol sym;
37     sym = DoLookupSymbol(
38             &GlobalIDs, name, !SEARCH_OUTER_TABLE);
39     if(sym == NULL){
40         AddSymbol(&GlobalIDs, (Symbol)p);
41     }else{
42         if(!IsCompatibleType(sym->ty, ty)){
43             Error(pcoord, "Incompatible with previous declaration");
44         }else{
45             sym->ty = CompositeType(ty, sym->ty);
46         }
47         return AddSymbol(Identifiers, sym);
48     }
49 }
50 return AddSymbol(Identifiers, (Symbol)p);
51 }

```

图 4.22 AddFunction() 和 AddVariable()

图 4.22 第 21 至 51 行的函数 AddVariable() 用于在当前符号表中添加一个变量名，第 25 至 27 行创建一个 variableSymbol 对象 p 并初始化。如果该变量处于全局作用域，或者是 static 变量，则通过图 4.22 第 29 至 30 行将其加入图 4.21 第 18 行的 Globals 链表中；而如果是局部变量，则在图 4.22 第 32 至 33 行将其加入当前函数的局部变量链表中。如果该变量位于局部作用域，且是通过 extern 关键字来声明的，我们在第 37 行调用 DoLookupSymbol() 函数，检查一下在全局符号表 GlobalIDs 中是否存在同名符号 sym，如果不存在，就在第 40 行把符号 p 加入 GlobalIDs；如果存在，就在第 42 至 46 行检查一下 p 跟已有的同名符号 sym 是否类型相容，之后在第 47 行将 sym 加入当前符号表 Identifiers 中。

## 4.2.5 成员选择运算符的语义检查

在 C 语言中，结构体和联合体被称为记录类型（RecordType），在形如 dt.a 和 ptr->a 的后缀表达式中，运算符 . 和 -> 被称为成员选择运算符。函数 CheckMemberAccess() 用于对这些表达式进行语义检查，与之相关的代码如图 4.23 所示。在表达式 dt.a 中，dt 和 a 相当于运算是 . 的两个操作数，dt 对应语法树结点的类型应是记录类型，图 4.23 第 8 行对此进行了检查；而在形如 ptr->a 的表达式中，ptr 对应结点的类型应为“指向记录”的指针，第 15 行的 if 语句对此进行了检查。图 4.23 第 4 行用于对成员选择运算符的左操作数进行语义检查。

结构体对象 dt 可当作左值，由此 dt.a 也可充当左值；但由于函数调用 GetData() 的返回值只存放在临时变量中，则 GetData().a 也不可充当左值。因此，图 4.23 第 11 行会根据运算符 “.” 的左操作数是否为左值来设置整个表达式 dt.a 或者 GetData().a 的 lvalue 属性。

对 ptr->a 来说，左操作数不必是左值，即使是个临时变量也可，例如在表达式 ((Data \*)ptr2)->a 中，强制转型后的结果 ((Data \*)ptr2) 就存放在一个临时变量中，但 ptr 或 ((Data \*)ptr2) 的值代表的是一个可寻址内存单元的地址，所以我们可以把 ptr->a 当作左值来用，第 19 行完成了这个设置。当然，即便表达式 dt.a 或者 ptr->a 对应的语法树结点的 lvalue 域为 1，在对 dt.a 或者 ptr->a 进行赋值时，我们还会检查一下其类型信息，看看是否有 const 限定符，如第 27 行的注释所示，如果定义 dt 变量时有 const 的限定，那么 dt.a 也相当于有 const 限定符，第 28 行的 Qualify() 函数用于添加这个类型限定符。第 21 行调用函数 LookupField() 检查一下结构体 Data 中是否有名字为 a 的域成员，结合图 2.18 来理解这个函数不会太难。如果在结构体的定义中找不到成员 a，则说明 dt.a 是非法的表达式，UCC 会在第 23 行报错。我们把

查询得到的关于域成员 a 的类型信息，通过第 29 行存放到 dt.a 表达式对应的语法树结点上。结构体 struct filed 描述了域成员的相关信息，我们在第 2 章的图 2.17 中介绍过这个结构体，这里不再重复。

```

1 static AstExpression CheckMemberAccess(AstExpression expr) {
2     Type ty;
3     Field fld;
4     expr->kids[0] = CheckExpression(expr->kids[0]);
5     if (expr->op == OP_MEMBER){// dt.a, GetData().a
6         expr->kids[0] = Adjust(expr->kids[0], 0);
7         ty = expr->kids[0]->ty;
8         if (! IsRecordType(ty)){
9             REPORT_OP_ERROR;
10        }
11        expr->lvalue = expr->kids[0]->lvalue;
12    }else{ // c->d, ((Data *) ptr)->a;
13        expr->kids[0] = Adjust(expr->kids[0], 1);
14        ty = expr->kids[0]->ty;
15        if (! (IsPtrType(ty) && IsRecordType(ty->bty))){
16            REPORT_OP_ERROR;
17        }
18        ty = ty->bty;
19        expr->lvalue = 1;
20    }
21    fld = LookupField(Unqual(ty), expr->val.p);
22    if (fld == NULL){
23        Error(&expr->coord, "struct or union member %s doesn't exist", expr->val.p);
24        expr->ty = T(INT);
25        return expr;
26    }
27    // const Data dt;          dt.a;
28    expr->ty = Qualify(ty->qual, fld->ty);
29    expr->val.p = fld;
30    expr->bitfld = fld->bits != 0;
31    return expr;
32 }
```

图 4.23 CheckMemberAccess()

图 4.24 给出了表达式 dt.a 在语法分析后和语义检查后所对应的语法树，对 op 域为 OP\_ID 的语法树结点的语义检查是由函数 CheckPrimaryExpression() 来完成的，在语义检查后，域成员 a 在结构体 struct Data 中的偏移量等信息由图中的 struct filed 对象保存，而结构体对象 dt 对应的符号则由 struct symbol 对象来存放。

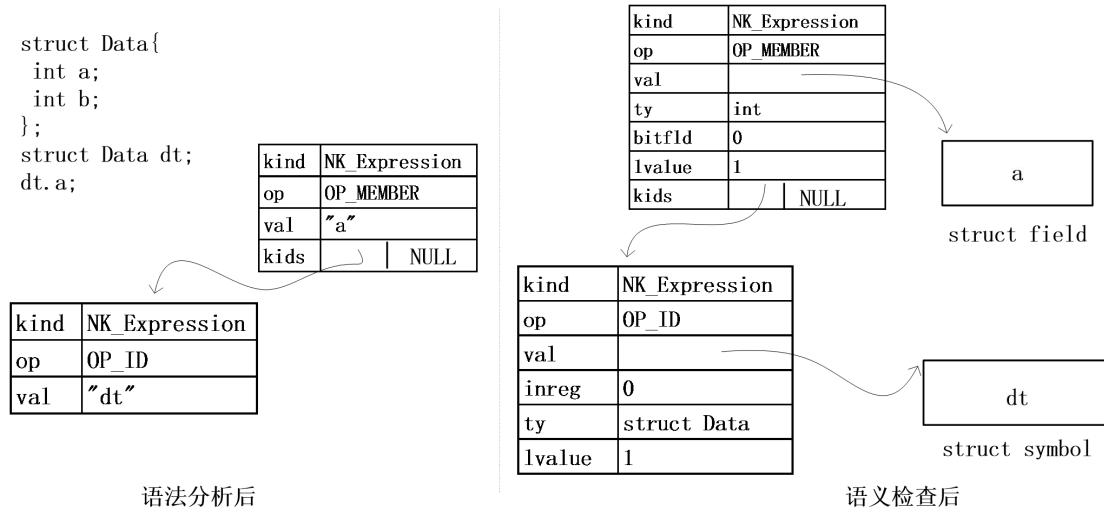


图 4.24 成员选择运算的语法树

接下来我们来看一下对“后加加”和“后减减”表达式的语义检查，例如 `a++` 和 `a--`。与其相关的代码如图 4.25 所示。由图 4.25 第 8 和 9 行的注释，我们可以知道，UCC 编译器会把 `a++` 和 `++a` 转换成 `a+=1` 来处理，而把 `a--` 和 `--a` 转换成 `a-=1` 来处理，第 6 行创建了一个语法树结点，第 10 行置其 `op` 域为 `OP_POSTINC` 或 `OP_PREINC`，即 `+=` 或者 `-=`，在第 15 行我们调用 `CheckExpression()` 来对赋值运算符 `+=` 或者 `-=` 进行语义检查。需要注意的是，结点 `expr` 的 `op` 域仍然保存着原有的 `OP_POSTINC` 或 `OP_POSTDEC` 等运算符，这样在生成代码时，我们能对 `a++`、`a--`、`++a` 或者 `--a` 进行区别，从而产生不同的代码。

```

1 static AstExpression TransformIncrement(AstExpression expr) {
2     AstExpression casgn;
3     union value val;
4
5     val.i[1] = 0; val.i[0] = 1;
6     CREATE_AST_NODE(casgn, Expression);
7     casgn->coord = expr->coord;
8     // a++, ++a -----> a += 1
9     // a--, --a -----> a -= 1
10    casgn->op = (expr->op == OP_POSTINC || expr->op == OP_PREINC)
11        ? OP_ADD_ASSIGN : OP_SUB_ASSIGN;
12    casgn->kids[0] = expr->kids[0];
13    casgn->kids[1] = Constant(expr->coord, T(INT), val);
14
15    expr->kids[0] = CheckExpression(casgn);
16    expr->ty = expr->kids[0]->ty;
17    return expr;
18 }
```

图 4.25 TransformIncrement()

至此，历经一番曲折，我们已经完成了对所有后缀表达式的语义检查。在后续章节中，我们会对一元表达式和二元表达式进行语义检查。沉舟侧畔千帆过，让我们继续扬帆前行。

## 4.2.6 相容类型

在前文对函数调用进行语义检查时，我们用函数 `CanAssign()` 来判断“能否把实参赋值给形参”，在函数 `CanAssign()` 中，我们又用宏 `IsCompatiblePtr()` 来判断两个指针是否相容。而如果指针类型 `T1 *` 和 `T2 *` 相容，则意味着类型 `T1` 和 `T2` 是相容的。UCC 编译器中

ucl\type.c 的函数 IsCompatibleType() 用于判断两个类型是否相容，与类型系统相关的数据结构请参见第 2.4 节中的各个示意图。在这一小节中，我们要对与 IsCompatibleType() 相关的几个函数做进一步讨论。我们先举个简单的例子来说明类型相容的概念，以下两个对 arr 的声明被视为相容，其出发点是遇到 int arr[] 时，我们只是暂时不知道其数组大小，稍后遇到 int arr[3] 时，可知道其数组大小为 3，因此以下对 arr 的两次声明对应的是内存中的同一个数组对象。需要注意的是，以下对于 a 的两次声明 int a 和 double a 被视为是冲突的，因为 int 占 4 字节而 double 占 8 字节，两者内部的数据组织形式也不一样，如果硬要规定把 a 当 double 来处理，那程序员在分析以下函数 f() 时，若没有注意到离得比较远的 double a，却发现在函数 f() 中 sizeof(a) 竟然是 8，那其脑门上不知要浮现出多少个问号。当然在进行赋值操作时，我们可以把整型的 b 赋值给 double 型的 c，这需要编译器进行隐式的类型转换，但不论如何转型，程序员使用变量名 b 和 c 时，编译器与程序员之间的约定仍旧是“b 对应内存中的一个 int 类型的变量，而 c 对应 double 类型的变量”。虽然在 UCC 编译器 exprchk.c 的个别地方用到了 IsCompatibleType() 函数，但这个函数主要还是用在 declchk.c 中，换言之，我们主要是在处理形如 int arr[] 和 int arr[3] 的声明时，用 IsCompatibleType() 函数来检查一下同名的两个声明的类型是否相容。

```

int arr[];
int arr[3];
int a;
int b; double c;
void f() {
    sizeof(a);
    c = b;
}
double a;

```

接下来，我们来分析一下 IsCompatibleType() 函数的代码，如图 4.26 所示。图 4.26 第 3 行用于判断 ty1 和 ty2 所是否指向同一个 struct type 对象，如果是则为相同类型的对象，自然就是彼此相容的；否则还需要进一步判断。

```

1 // 判断类型 ty1 与 ty2 是否相容
2 int IsCompatibleType(Type ty1, Type ty2) {
3     if (ty1 == ty2)
4         return 1;
5     if (ty1->qual != ty2->qual)
6         return 0;
7     ty1 = Unqual(ty1);
8     ty2 = Unqual(ty2);
9     //if (ty1->categ == ENUM && ty2 == ty1->bty ||
10 //    ty2->categ == ENUM && ty1 == ty2->bty) {
11 //    return 1;
12 //}
13     if (ty1->categ != ty2->categ)
14         return 0;
15     switch (ty1->categ) {
16     case POINTER: // recursive check ty1->bty and ty2->bty
17         return IsCompatibleType(ty1->bty, ty2->bty);
18     case ARRAY: // int arr[]; int arr[3]
19         return IsCompatibleType(ty1->bty, ty2->bty) &&
20             (ty1->size == ty2->size || ty1->size == 0 || ty2->size == 0);
21     case FUNCTION:
22         return IsCompatibleFunction((FunctionType)ty1, (FunctionType)ty2);
23     default: // STRUCT/UNION

```

```

24     return ty1 == ty2;
25 }
26 }
```

图 4.26 IsCompatibleType()

在 UCC 编译器中，我们可用图 4.26 第 3 行的条件来判断整型和浮点型，例如对 int a 和 int b 而言，由于 UCC 编译器用同一个 struct type 对象来表示在源代码出现的所有 int 类型，所以 a 和 b 的类型信息都存放在 T(INT) 所指向的 struct type 对象中；但对 int c 和 char d 而言，c 和 d 分别对应 T(INT) 和 T(CHAR)，它们指向两个不同的 struct type 对象，因此，图 4.26 第 3 行的条件不成立，即 int 和 char 不是相容的类型。再次强调的是，我们可以进行 int 型的 c 和 char 型的 d 之间的相互赋值，但是由于 int 和 char 是不相容的，在同一个作用域中，“同时声明 char c 和 int c” 则是错误的。UCC 编译器用以下数组 Types 来表示整型和浮点型等基本类型。

```
#define T(categ) (Types + categ)
struct type Types[VOID - CHAR + 1];
```

而 const int 和 int 也被视为不相容的类型，两者在限定符上不一致，图 4.26 第 5 行的条件用于此目的。而 int \* 和 struct Data 显然风马牛不相及，一个是指针类别，一个是结构体类别，第 13 行的 if 条件用于对此进行判断。对于形如 T1 \* 和 T2 \* 的类型来说，两者都是指针类别，我们就在第 17 行递归地判断 T1 和 T2 是否相容。而对于数组类型 T1 [m] 和 T2[n] 来说，两者相容的条件是“T1 和 T2 相容”并且“m 和 n 大小一样，或者两者中有一个为 0”，例如 int [3] 和 int [] 是相容的，图 4.26 第 18 至 20 行完成这样的判别。对函数类型来说，我们还需要递归地检查两个函数的对应参数和返回值类型的相容性，这通过在第 22 行调用 IsCompatibleFunction() 函数来处理。而对于结构体和联合体来说，我们通过第 24 行的条件来检查 ty1 和 ty2 是否指向同一个 struct type 对象。例如，UCC 编译器会为以下结构体 struct Data1 和 struct Data2 各建立一个 struct type 对象，用来存放它们的类型信息，这意味着，即使这两个结构体的所有成员的名称和类型都一样，但 struct Data1 和 struct Data2 是不相容的。

```

struct Data1{
    int a;
    double b;
};

struct Data2{
    int a;
    double b;
};
```

对于枚举类型和 int 来说，如果要把这两者视为相容的，则删去图 4.26 第 9 至 12 行的注释号 “//” 即可。枚举类型可视为范围受限的整型。按 ansi.c.txt 第 3.5.2.2 节的说明，这两者是相容的。但是 GCC 和 Clang 却视之为不相容类型，会对以下声明进行报错。保留图 4.26 第 9 至 12 行的注释号，UCC 编译器也可以对以下声明报错 “error:Incompatible with previous definition”。从类型安全的角度来考虑，“视 enum 和 int 为不相容类型” 是更妥当的，毕竟以下 enum Color 代表的取值范围实际上是 {0,1,2}，而 int 代表的取值范围则为 {..., -1, 0, 1, 2, 3, ...}。

```
int a;      // compatible in UCC, not in GCC/Clang
enum Color{RED, GREEN, BLUE} a;
```

现在，让我们来分析一下用于判断两个函数类型是否相容的 IsCompatibleFunction()，其代码如图 4.27 所示。我们主要从以下几个方面来判断：函数的返回值、函数的参数、是否变参、是新式风格的函数还是旧式风格的函数。

```

1 static int IsCompatibleFunction(
2         FunctionType fty1, FunctionType fty2){
```

```

3   Signature sig1 = fty1->sig;
4   Signature sig2 = fty2->sig;
5   Parameter p1, p2;
6   int parLen1, parLen2;
7   int i;
8   if (! IsCompatibleType(fty1->bty, fty2->bty))
9     return 0;
10  if (! sig1->hasProto && ! sig2->hasProto){
11    return 1;
12  } else if (sig1->hasProto && sig2->hasProto){
13    parLen1 = LEN(sig1->params);
14    parLen2 = LEN(sig2->params);
15    if ((sig1->hasEllipsis ^ sig2->hasEllipsis)
16        || (parLen1 != parLen2)){
17      return 0;
18    }
19    // recursive check parameters
20    for (i = 0; i < parLen1; ++i){
21      p1 = (Parameter)GET_ITEM(sig1->params, i);
22      p2 = (Parameter)GET_ITEM(sig2->params, i);
23      if (! IsCompatibleType(p1->ty, p2->ty))
24        return 0;
25    }
26    return 1;
27  }
28  else if (! sig1->hasProto && sig2->hasProto){
29    sig1 = fty2->sig;
30    sig2 = fty1->sig;
31    goto mix_proto;
32  } else{
33 mix_proto:
34   parLen1 = LEN(sig1->params);
35   parLen2 = LEN(sig2->params);
36   if (sig1->hasEllipsis)
37     return 0;
38   if (parLen2 == 0){
39     FOR_EACH_ITEM(Parameter, p1, sig1->params)
40       if (! IsCompatibleType(Promote(p1->ty), p1->ty))
41         return 0;
42   ENDFOR
43   return 1;
44  } else if (parLen1 != parLen2){
45    return 0;
46  } else{
47    for (i = 0; i < parLen1; ++i){
48      p1 = (Parameter)GET_ITEM(sig1->params, i);
49      p2 = (Parameter)GET_ITEM(sig2->params, i);
50      if (! IsCompatibleType(p1->ty, Promote(p2->ty)))
51        return 0;
52    }
53    return 1;
54  }
55 }
```

---

56 }

图 4.27 IsCompatibleFunction()

图 4.27 第 8 行用于判断两个函数的返回值类型是否相容，如果返回值类型相容且两个函数都是旧式风格的函数（不具有原型），我们就从第 11 行返回。对于新式风格的函数，其参数要成为其接口的一部分，我们要判断一下两者参数个数是否相同，且参数类型是否相容，第 20 至 25 行的 for 循环用于此目的。当然，如果其中一个是变参函数，而另一个不是变参函数，这实际上会导致两个声明对参数个数的约束是不一样的，例如以下对 f 的两次声明的是冲突的。图 4.27 第 15 行的 if 条件用于对此进行判断。第 12 至 27 行用于处理两个函数类型都是新式函数的情况。

```
int f(int a);
int f(int a, ...);
```

而如果一个函数是新式函数，另一个是旧式函数，为了处理方便，我们希望执行到第 33 行时，sig1 能指向新式函数的参数列表，而让 sig2 指向旧式函数的参数列表，图 4.27 第 28 至 31 行进行必要的互换，从而达到这个目的。图 4.27 第 33 至 55 行的代码是根据 C89 标准文档 ansi.c.txt 第 3.5.4.3 节的语义规则来编写的。对以下两个声明来说，由于变参函数 void f(float a,...) 是新式风格函数的声明，对有名的参数 a 不会进行实参提升的操作，对于无名参数，则会进行实参提升操作；由于 void f() 是旧式风格函数，所有的实参都会进行实参提升的操作。

```
void f(float a,...);
void f();
```

如果执行函数调用 f(1.0f,2.0f)，对于第一个实参 1.0f 而言，若按声明 void f(float a,...) 的约束，我们不需要进行实参提升，真正入栈的参数就是 float 型的 1.0f；而按 void f() 的要求，我们需要进行实参提升，真正入栈的是 double 型的 1.0。这是两个矛盾的要求，所以 UCC 把这样的两个函数类型当作不相容的类型。

对于声明 void g(int, float) 和 g() 而言，当我们执行函数调用 g(3,4.0f) 时，第 2 个参数 4.0f 会出现同样的问题。但对声明 void h(int,int) 和 h() 来说，当我们执行函数调用 h(5,6) 时，进行实参提升后，真正入栈的参数仍然是两个 int 型的参数，这不会出现冲突，我们可以视之为相容类型。图 4.27 第 36 至 43 行对这些情况进行判断。

而对于以下代码来说，按旧式风格的函数 h(a,b) 的要求，当我们执行函数调用 h(1.0,2.0) 时，实参提升后的入栈参数仍旧是 2 个 double 型的浮点数；但按照 void h(int,int) 的约束，我们需要先把 double 实参隐式地强制转型为 int 型整数，再把转型后得到的 2 个 int 型整数入栈，实际执行的函数调用相当于 h((int)1.0,(int)2.0)。因此，“void h(a,b)double a,b;” 和 “void h(int,int);” 也是冲突的。图 4.27 第 47 至 53 行用于对此进行判断。

```
void h(a,b)double a,b;{
}
void h(int,int);
```

需要再次强调的是，我们不应去使用旧式风格的函数声明，应编写新式风格的函数，这样 C 编译器会对函数参数进行更严格的检查，旧式风格的函数所带来的问题我们在第 2 章时已举过例子。但对 C 编译器而言，只好背上这个历史的包袱。

当面对如下相容的函数声明时，我们需要对这两个类型进行综合，合成出一个与这两者都相容的函数类型，这相当于“求两个整数的最大公约数”的过程。合成后产生的类型就相当于原先两个类型的最大公约数。

```
int f(int (*)(), double (*)[3]); // f1
int f(int (*)(char *), double (*)[]); // f2
```

为了表述方便，我们称第一个 f 为 f1，第二个 f 为 f2。函数 f1 的第 1 个参数的类型为 int(\*)(), 这是一个“指向旧式函数类型 int ()”的指针，而函数 f2 的第 1 个参数为 int (\*)(char

\*)，这是一个“指向新式函数类型 int(char \*)”的指针，这两个指针类型是相容的。函数类型 int(char \*)对参数的要求更为严格，所以它们的“最大公约数”为 int(char \*)。同理，f1 的第 2 个参数和 f2 的第 2 个参数的“最大公约数”为 double(\*)[3]，即“指向 double[3]”的指针类型。由此，我们得到 f1 和 f2 的合成类型为：

```
int f(int (*)(char *), double (*)[3]);
```

用于合成新类型的函数 CompositeType()如图 4.28 所示，由第 2 行的断言可知，只有对相容的两个类型，我们才能求其合成类型。对于形如 const T1 \* 和 const T2 \* 的两个相容类型，我们递归地去求 T1 和 T2 的合成类型，不妨记为 T3，由此 const T1 \* 和 const T2 \* 的合成类型为 const T3 \*，第 8 至 9 行的代码完成了这个工作。而对于形如 int [3] 和 int [] 的两个数组类型，我们为之合成的类型即为 int [3]，第 10 和第 11 行完成这个工作。而对于函数类型，我们需要在第 17 行合成返回值类型，然后在第 20 至 27 行递归地对各个参数类型依次进行合成。

```

1 Type CompositeType(Type ty1, Type ty2) {
2     assert(IsCompatibleType(ty1, ty2));
3     if (ty1->categ == ENUM)
4         return ty1;
5     if (ty2->categ == ENUM)
6         return ty2;
7     switch (ty1->categ) {
8     case POINTER: // const T1 *, const T2 * ---> const T3 *
9         return Qualify(ty1->qual, PointerTo(CompositeType(ty1->bty, ty2->bty)));
10    case ARRAY: // int arr[3], int arr[] ---> int arr[3]
11        return ty1->size != 0 ? ty1 : ty2;
12    case FUNCTION:
13        {
14            FunctionType fty1 = (FunctionType)ty1;
15            FunctionType fty2 = (FunctionType)ty2;
16            // return type
17            fty1->bty = CompositeType(fty1->bty, fty2->bty);
18            // parameters
19            if (fty1->sig->hasProto && fty2->sig->hasProto) {
20                Parameter p1, p2;
21                int i, len = LEN(fty1->sig->params);
22                for (i = 0; i < len; ++i) {
23                    p1 = (Parameter)GET_ITEM(fty1->sig->params, i);
24                    p2 = (Parameter)GET_ITEM(fty2->sig->params, i);
25                    p1->ty = CompositeType(p1->ty, p2->ty);
26                }
27                return ty1;
28            }
29            return fty1->sig->hasProto ? ty1 : ty2;
30        }
31    default:
32        return ty1;
33    }
34 }
```

图 4.28 CompositeType()

在 ucl\type.c 中，还有一个用于求两个实数类型的“最小公倍数”的函数 CommonRealType()。当我们执行以下代码，要进行 a+c 运算时，我们需要把 int 型的 c 隐式地转型为 double 型，然后与 double 型的 a 进行浮点数的加法。按前面的讨论，int 和 double

型变量所占内存大小不一样，数据组织格式也不一样，是不相容的类型。由于 double 占 8 个字节，为了最大限度地保留数据精度，我们很自然地会选择把 c 隐式地转换为 double 类型，这其实相当于求 int 和 double 型的“最小公倍数”。

```
double a,b;
int c;
b = a+c;
```

C89 标准文档 ansi.c.txt 第 3.2.1.5 节规定了用于算术类型的转换规则，由此编写的代码如图 4.29 所示。图中第 2 行的参数 ty1 和 ty2 都是算术类型，即浮点数或者整型。当 ty1 或 ty2 有一个为 long double，则公共类型是能保留更大精度的 long double；当 ty1 或者 ty2 有一个为 double，则公共类型为 double；而当 ty1 或者 ty2 有一个为 float，则公共类型为 float。第 3 至 8 行依次完成了这些判断，接下来的第 9 至 29 行则用于处理 ty1 和 ty2 都是整型的情况，如 char、short、int 或者 long 等。

```
1 // 公共类型
2 Type CommonRealType(Type ty1, Type ty2) {
3     if (ty1->categ == LONGDOUBLE || ty2->categ == LONGDOUBLE)
4         return T(LONGDOUBLE);
5     if (ty1->categ == DOUBLE || ty2->categ == DOUBLE)
6         return T(DOUBLE);
7     if (ty1->categ == FLOAT || ty2->categ == FLOAT)
8         return T(FLOAT);
9     // neither ty1 nor ty2 is floating number.
10    ty1 = ty1->categ < INT ? T(INT) : ty1;
11    ty2 = ty2->categ < INT ? T(INT) : ty2;
12    if (ty1->categ == ty2->categ)
13        return ty1;
14    // ty1 and ty2 have the same sign
15    if ((IsUnsigned(ty1) ^ IsUnsigned(ty2)) == 0)
16        return ty1->categ > ty2->categ ? ty1 : ty2;
17    // Their signs are different.
18    // Swap ty1 and ty2, then we treat ty1 as Unsigned, ty2 as signed later.
19    if (IsUnsigned(ty2)){
20        Type ty;
21        ty = ty1;
22        ty1 = ty2;
23        ty2 = ty;
24    }
25    if (ty1->categ >= ty2->categ)
26        return ty1;
27    if (ty2->size > ty1->size)
28        return ty2;
29    return T(ty2->categ + 1);
30 }
```

图 4.29 CommonRealType()

在 UCC 编译器中，参与算术运算的整型如果低于 int 型（例如 char 或者 short），则会被先提升为 int 型。在 16 位和 32 位机器中，int 型实际上反映了数据寄存器的大小，我们总是希望 CPU 的操作数会尽可能地出现在寄存器中。第 10 至 11 行完成了这个类型提升操作。当两个整型的符号位一样，即都为无符号整型或者都是有符号整型，我们就取“要占更大内存空间的类型”为公共类型，第 15 至 16 行用于此操作。当两个整型的符号位不一样时，为了处理上的方便，第 19 至 24 行进行必要的交换，此后，ty1 就对应无符号整型，而 ty2 则对应有符号整型。第 25 至 29 行用于选择占更大内存的类型，如果两者所占内存一样，我们

就取无符号整型为公共类型。

#### 4.2.7 一元表达式的语义检查

在这一小节中，我们来讨论一元运算符表达式的语义检查，与其相关的代码如图 4.30 所示。对于“前加加”和“前减减”运算符而言，我们采取的策略跟处理“后加加”和“后减减”一样，都是将`--a`转换为`a = 1`，而将`++a`转换为`a += 1`，所以图 4.30 第 5 行调用的函数，就是我们在讨论后缀表达式语义检查时介绍过的函数`TransformIncrement()`。对于形如`+a`或`-a`的表达式，我们需要检查一下`a`是否为算术类型，由于表达式`+a`的结果存放在一个临时变量中，所以`+a`是个右值，此时表达式`+a`的值即为`a`，第 14 行的`Adjust()`函数把`a`对应语法树结点视为右值；而对于`-a`，如果`a`对应一个常数，比如`(3.0+4.0)`，那我们在编译时就可以进行`-(3.0+4.0)`的计算从而得到`-7.0`，第 18 行的`FoldConstant()`完成了这个常量折叠的操作。若操作数`a`的类型小于`int`型，则我们通过第 16 行的`DoIntegerPromotion()`进行整型提升，由第 17 行我们置表达式`+a`的类型为“子表达式`a`的类型”。对于按位取反运算符`~`来说，按 C 的语义规则，表达式`~a`中操作数`a`的类型应为整型，第 21 至 28 行的代码不难理解，这里不再啰嗦。

```

1 static AstExpression CheckUnaryExpression(AstExpression expr) {
2     Type ty;
3     switch (expr->op) {
4         case OP_PREINC: case OP_PREDEC: // --a, ++a
5             return TransformIncrement(expr);
6         case OP_ADDRESS: // &a
7             ...
8             break;
9         case OP_DEREF: // *ptr
10        ...
11        break;
12        case OP_POS: // +a
13        case OP_NEG: // -a
14            expr->kids[0] = Adjust(CheckExpression(expr->kids[0]), 1);
15            if (IsArithType(expr->kids[0]->ty)) {
16                expr->kids[0] = DoIntegerPromotion(expr->kids[0]);
17                expr->ty = expr->kids[0]->ty;
18                return expr->op == OP_POS ? expr->kids[0] : FoldConstant(expr);
19            }
20            break;
21        case OP_COMP: // ~a
22            expr->kids[0] = Adjust(CheckExpression(expr->kids[0]), 1);
23            if (IsIntegType(expr->kids[0]->ty)) {
24                expr->kids[0] = DoIntegerPromotion(expr->kids[0]);
25                expr->ty = expr->kids[0]->ty;
26                return FoldConstant(expr);
27            }
28            break;
29        case OP_NOT: // !a
30            expr->kids[0] = Adjust(CheckExpression(expr->kids[0]), 1);
31            if (IsScalarType(expr->kids[0]->ty)) {
32                expr->ty = T(INT);
33                return FoldConstant(expr);
34            }

```

```

35     break;
36 case OP_SIZEOF: // sizeof(a), sizeof(int)
37     if (expr->kids[0]->kind == NK_Expression) {
38         expr->kids[0] = CheckExpression(expr->kids[0]);
39         if (expr->kids[0]->bitfld) {
40             goto err;
41         }
42         ty = expr->kids[0]->ty;
43     } else{
44         ty = CheckTypeName((AstTypeName)expr->kids[0]);
45     }
46     if (IsFunctionType(ty) ||
47         IsIncompleteType(ty, !IGNORE_ZERO_SIZE_ARRAY)) {
48         goto err;
49     }
50     expr->ty = T(UINT);
51     expr->op = OP_CONST;
52     expr->val.i[0] = ty->size;
53     return expr;
54 case OP_CAST: // (int)a
55     return CheckTypeCast(expr);
56 default:
57     assert(0);
58 }
59 err:
60 REPORT_OP_ERROR;
61 }

```

图 4.30 CheckUnaryExpression()

而对于逻辑非运算符!来说，与之对应的代码在图 4.30 第 29 至 35 行。表达式!a 中的子表达式 a 的类型应是标量类型，即整型、浮点型和指针类型，而数组类型通常被视为向量类型，按数学上的定义，向量相当于是多维空间中的坐标(x<sub>1</sub>,x<sub>2</sub>,...,x<sub>n</sub>)，而标量只是一维空间上的坐标 x。结构体类型也不是标量类型。图 4.30 第 31 行的 IsScalarType() 用于判断某类型是否为标量。当然，逻辑非!a 的运算结果应为布尔型，但 C 语言中并没有布尔型，而是用 0 和非 0 来表示，这对应的是 int 型，第 32 行我们置!a 对应语法树结点为 int 型。

对于一元运算符 sizeof 而言，图 4.30 第 37 至 42 行的代码用于处理形如 sizeof(a+b) 的一元表达式，此时 sizeof 运算符的操作数是一个表达式，但按 C 标准的规定，结构体中的位域成员不可以作为 sizeof 的操作数，第 39 至 41 行对此进行了检查；而第 43 至 45 行的代码用于处理形如 sizeof(int \*) 的表达式，此时 sizeof 的操作数为一个类型名，第 44 行的 CheckTypeName() 用于对类型名进行语义检查，我们会在讨论 declchk.c 时对这个函数进行分析，其函数返回值是一个指向 struct type 对象的指针，该 struct type 对象中就包含了类型信息，例如该类型的变量要占多大内存。整个 sizeof 表达式应是个编译时的无符号整数常量，第 50 至 52 行完成了这些设置。

接下来我们来看一下图 4.30 第 55 行的 CheckTypeCast() 函数，该函数用于对形如 (int)a 这样的强制类型转换表达式进行语义检查，其代码如图 4.31 所示。在图 4.31 第 7 行，我们调用 CheckTypeName() 函数来对 (int) a 中的类型名 int 进行语义检查，从而得到与类型名对应的 struct type 对象；第 9 行则通过调用 Adjust() 函数来对操作数 a 进行必要的类型调整。第 11 至 14 行，我们引用了来自 C 标准文档 ansi.c.txt 中的一段语义规则，由此可知，形如 (T) expr 的强制类型转换表达式中，T 的类型和 expr 的类型都得是标量类型，不可以是数组类型或结构体类型，当然 T 可以是 void，第 16 至 19 行的代码实现了这些语义规则。第 20 行

调用 Cast 函数来进行类型转换，我们已在之前的章节中分析过 Cast() 函数。

```

1 // (int) a
2 // expr:          OP_CAST
3 // kids[0]:       int
4 // kids[1]:       a
5 static AstExpression CheckTypeCast(AstExpression expr) {
6     Type ty;
7     ty = CheckTypeName((AstTypeName)expr->kids[0]);
8     //
9     expr->kids[1] = Adjust(CheckExpression(expr->kids[1]), 1);
10 //*****
11 see asn1.c.txt
12         Unless the type name specifies void type, the type name shall
13 specify qualified or unqualified scalar type and the operand shall
14 have scalar type.
15 ****
16 if (! (BothScalarType(ty, expr->kids[1]->ty) || ty->categ == VOID)) {
17     Error(&expr->coord, "Illegal type cast");
18     return expr->kids[1];
19 }
20 return Cast(ty, expr->kids[1]);
21 }
```

图 4.31 CheckTypeCast()

在图 4.30 中，我们还需要在第 6 行检查形如 `&a` 的取地址运算，或者在第 9 行处理形如 `*ptr` 的“提领”操作，由于这些代码相对复杂，我们没有在图 4.30 中给出。接下来我们先举个例子来说明一下形如 `*ptr` 的提领运算，如图 4.32 所示。

```

1 // hello.c C 源代码
2 int arr[3][4];
3 typedef int (*ArrPtr)[4];
4 ArrPtr ptr = &arr[0];
5 int * ptr1 = &arr[0][0];
6 int ** ptr2 = &ptr1;
7 int main(int argc,char * argv[]){
8     **arr = 1;
9     **ptr = 2;
10    **ptr2 = 3;
11    return 0;
12 }
13 // hello.ast 抽象语法树
14 function main
15 {
16     (= ([] ([] arr
17                 0)
18                 0)
19                 1)
20     (= ([] ([] ptr
21                 0)
22                 0)
23                 2)
24     (= (* (* ptr2))
25         3)
26     (ret 0)
27 }
28
29
30 // UCC 编译器生成的中间代码
31 // hello.uil
32 function main
33 arr[0] = 1;
34 *ptr = 2;
35 t3 :*ptr2;
36 *t3 = 3;
37 return 0;
38 ret
39 // hello.s 汇编代码
40 movl $1, arr+0
41 movl ptr, %eax
42 movl $2, (%eax)
43 movl ptr2, %eax
44 movl (%eax), %ecx
45 movl $3, (%ecx)
```

图 4.32 提领运算

虽然在图 4.32 第 8 至 10 行，我们都是用 `*` 运算符来表达 C 语言的提领运算。但在生成的汇编代码中是否需要进行间接寻址，则要看操作数的类型。换言之，语法上相似的表达式，

如`**arr`、`**ptr`和`**ptr2`，由于`arr`、`ptr`和`ptr2`等操作数在类型上的差别，导致其经语义检查后生成的语法树也有较大差别，其内存寻址模式也是不一样的。

例如对于图 4.32 第 8 行的`**arr = 1`而言，我们不需要进行任何的间接寻址，与其对应的汇编指令为第 40 行的“`movl $1, arr+0`”，在链接时，`arr`就相当于一个地址常量。第 8 行的赋值运算的左操作数`**arr`，对应的内存地址就是`arr+0`。所以在`**arr`经语义检查后得到的语法树`([] ([] arr 0) 0)`中，并没有出现提领运算`*`，而是数组索引运算符`[]`。在生成中间代码时，对于数组索引运算符，我们实际上进行的运算主要是加法，此处只要进行`arr+0+0`，就可得到左操作数的地址。做这些决策的依据是：`arr`对应的类型是数组类型`int [3][4]`，而不是指针类型。

由图 4.22 第 6 行我们可知，`ptr2`是个指针类型`int **`，并且不是指向数组的指针。对于图 4.32 第 10 行的`**ptr2 = 3`来说，我们需要进行两次的间接寻址，才能得到左操作数的内存地址，其汇编代码如图 4.32 第 43 至 45 行所示。第 43 行把`ptr2`对应内存单元的内容送到寄存器`eax`中，第 44 行取寄存器`eax`所指向内存单元的内容，存到寄存器`ecx`中，这里我们做了一次寄存器间接寻址；第 45 行把立即数 3 存到`ecx`所指向的内存单元中，这里我们又做了一次寄存器间接寻址。寄存器间接寻址在 AT&T 汇编代码上的语法特征形如`(%eax)`，表示寄存器`eax`中存放的是某个内存单元的地址。语义检查后，与`**ptr2`对应的语法树为第 24 行的`(*(* ptr2))`，此时在语法树上出现的`*`运算符，表示我们确实需要进行提领运算，汇编指令通过间接寻址来实现提领运算。

对于图 4.32 第 9 行的`**ptr = 2`，由第 3 和第 4 行的声明我们可知，`ptr`的类型是“指向数组`int[4]`”的指针。语义检查后，我们为`**ptr`生成的语法树是`([] ([] ptr 0) 0)`，这和`**arr`经语义检查后的语法树`([] ([] arr 0) 0)`相似。由于`ptr`是指向数组的指针类型，而`arr`是数组类型，我们在中间代码生成时，会为`([] ([] ptr 0) 0)`和`([] ([] arr 0) 0)`生成不同的代码。因此，与`**ptr`对应的汇编代码为第 41 和 42 行，第 41 行的“`movl ptr,%eax`”用于把`ptr`对应内存单元的内容送到寄存器`eax`中，第 42 行的“`mov $2 (%eax)`”通过寄存器间接寻址，把立即数 2 送入`eax`所指向的内存单元，这里我们做了一次间接寻址操作。

简而言之，对于形如`*p`的表达式，“如果`p`是数组类型，或者`*p`是数组类型”，语法分析后我们得到的语法树为`(* p)`，但在语义检查后，我们为`*p`构造的语法树为`([] p 0)`。稍作推广一下，对于形如`*(p+i)`的表达式，语法分析后对应的语法树为`(* (+ p i))`，“如果`p`是数组类型，或者`*(p+i)`是数组类型”，我们可将其转换为`p[i]`来处理。在语义检查后，我们为之构造的语法树为`([] p k)`，其中`k`为`i*sizeof(*p)`。例如对`int arr[3][4]`来说，表达式`*(arr+1)`经语义检查后对应的语法树为`([] arr 16)`，其中 16 源于`1*sizeof(*arr)`，即`1*sizeof(int[4])`。在函数`CheckUnaryExpression()`中，与`dereference`运算符相关的代码如图 4.33 所示。

```

1 static AstExpression CheckUnaryExpression(
2                                     AstExpression expr) {
3     Type ty;
4     switch (expr->op) {
5         //略
6     case OP_DEREF: // *p
7         expr->kids[0] =
8             Adjust(CheckExpression(expr->kids[0]), 1);
9         ty = expr->kids[0]->ty;
10        if (expr->kids[0]->op == OP_ADDRESS) {
11            // *(&a) ---> a
12            expr->kids[0]->kids[0]->ty = ty->bty;
13            return expr->kids[0]->kids[0];
14        } else if (expr->kids[0]->op == OP_ADD &&
15                  (ty->bty->categ == ARRAY)

```

```

16             || expr->kids[0]->kids[0]->isarray) ) {
17         // * (arr+3) ---> arr[3]
18         expr->kids[0]->op = OP_INDEX;
19         expr->kids[0]->ty = ty->bty;
20         expr->kids[0]->lvalue = 1;
21         return expr->kids[0];
22     }
23     //
24     if (IsPtrType(ty)) {
25         expr->ty = ty->bty;
26         // void f(void){}
27         if (IsFunctionType(expr->ty)) {
28             // (*f) () ---> f()
29             return expr->kids[0];
30         }
31         // *ptr ---> ptr[0]
32         if(expr->ty->categ == ARRAY
33             || expr->kids[0]->isarray) {
34             union value val;
35             val.i[0] = val.i[1] = 0;
36             expr->kids[1] =
37                 Constant(expr->coord, T(INT), val);
38             expr->op = OP_INDEX;
39         }
40         expr->lvalue = 1;
41         return expr;
42     }
43     break;
44 //略
45 }
46 }
```

图 4.33 CheckUnaryExpression()\_OP\_DEREF

图 4.33 第 14 至 22 行的代码用于把表达式 $*(arr+i)$ 对应的语法树 $(*(+ arr i))$ , 转换为 $([] arr k)$ 来处理, 由于我们是后序遍历语法树, 所以由 $i*sizeof(*arr)$ 得到 $k$ 的计算会在访问运算符“+”对应结点时进行, 而访问运算符“+”对应结点之后, 我们才会去访问其父结点, 即此处的\*运算符结点。第 31 至 39 行的代码则用于把表达式 $*ptr$ 的语法树 $(* ptr)$ 转换为 $([] ptr 0)$ 。而第 10 至 13 行的代码则用于把形如 $*(\&a)$ 的表达式转换为 $a$ , 第 27 至 30 行的代码则用于把形如 $(*f)()$ 的函数调用转换为 $f()$ 。

接下来, 我们来看一下取地址运算符&。我们结合一个简单的例子来说明一下数组名在不同场合的微妙的语义区别。例如, 对数组 int arr[3][4]而言, 在符号表中存放的标识符 arr 的类型为数组类型 int [3][4]。在表达式 $(arr+1)$ 中, arr 对应的语法树结点的类型会被调整为调整为 int (\*)[4], 这就是我们平时在 C 语言中所说的“数组名 arr 代表的是数组第 0 个元素 arr[0]的首地址”。严格说来, 这句话并不够准确, 在符号表中, 符号 arr 的类型始终都是数组类型 int [3][4], 至于 arr 对应语法树结点的类型, 则要看 arr 所处的表达式上下文, 在 $(arr+1)$ 中, arr 结点的类型会被调整为 int (\*)[4]。但是在表达式 $\&arr$ 中, 按 C 的语义, 我们不需要对 arr 结点的类型进行调整, 即不需要调用 UCC 编译器中的 Adjust()函数, 所以表达式 $\&arr$ 的类型为 int (\*)[3][4], 即指向“数组 int [3][4]”的指针类型。上机做个小实验就会发现 arr+1 和 $\&arr+1$ 的值是不一样的。

```

printf("%p %p %p \n", arr, arr+1, &arr+1);
//实验结果    804a060      804a070          804a090
```

表达式 arr+1 中 arr 结点的类型被调整为 int (\*)[4]。按指针运算的语义，T \* 类型的指针 ptr 进行(ptr+1)的运算，其含义是指向下一个 T 类型的对象，这意味着在汇编代码层次，真正执行的加法运算为(ptr + sizeof(T))。此处类型 T 为 int[4]，而 sizeof(int[4]) 为 16，写成十六进制即为 0x10。而对表达式&arr+1 中的 arr 而言，其所处的子表达式为&arr，子表达式&arr 的类型为 int (\*)[3][4]，此处 T 为 int [3][4]，sizeof(T) 为 48，对应十六进制的 0x30。若数组 arr 的首地址为 0x804a060，则 arr+1 的值为 0x804a070，而&arr+1 的值为 0x804a090。

与取地址运算符&其相关的代码如图 4.34 所示。可以看到，在图 4.34 第 6 行中，我们并没有调用 Adjust() 函数进行类型调整。第 8 至 11 行用于把形如&(\*ptr) 的表达式转换为 ptr 来处理，由于&(\*ptr) 是个右值，所以我们要把转换后得到的 ptr 结点也当右值来处理，因此在第 10 行置 lvalue 域为 0。而第 12 至 18 行则用于把&arr[i] 的语法树转换为(+ arr k) 来处理，由于是后序遍历，由 i 进行 i\*sizeof(\*arr) 从而得到 k 的运算，已经在处理子树 arr[i] 时完成。只有操作数 a 是左值时，我们才能进行&a 的运算，左值意味着对 C 程序员而言，该单元是可寻址的。如果&a 中的结点 a 对应的是函数名，由于我们在 CheckPrimaryExpression() 函数中，已把该结点的 lvalue 域置为 0，所以这里需要在第 19 行进行特殊判断一下。当然，按 C 的语义，C 程序员不可以对结构体中的位域成员和被声明为 register 的变量进行取地址运算，第 20 行对此进行了判断。由于表达式&a 是右值，所以第 23 行把&a 对应结点的 lvalue 域置为 0，第 24 行完成了对其类型的设置。

```

1 //对一元表达式进行语义检查
2 static AstExpression CheckUnaryExpression(AstExpression expr) {
3     Type ty;
4     switch (expr->op) {
5         case OP_ADDRESS:    // &a
6             expr->kids[0] = CheckExpression(expr->kids[0]);
7             ty = expr->kids[0]->ty;
8             if (expr->kids[0]->op == OP_DEREF) {
9                 //&(*ptr) ----> ptr
10                expr->kids[0]->kids[0]->lvalue = 0;
11                return expr->kids[0]->kids[0];
12            }else if (expr->kids[0]->op == OP_INDEX) {
13                // &arr[i] ---> (+ arr k)
14                expr->kids[0]->op = OP_ADD;
15                expr->kids[0]->ty = PointerTo(ty);
16                expr->kids[0]->lvalue = 0;
17                return expr->kids[0];
18            }
19            else if (IsFunctionType(ty) ||
20                  (expr->kids[0]->lvalue &&
21                   !expr->kids[0]->bitfld && !expr->kids[0]->inreg)) {
22                // &arr
23                expr->lvalue = 0;
24                expr->ty = PointerTo(ty);
25                return expr;
26            }
27            break;
28        //略
29    }
30 }
```

图 4.34 CheckUnaryExpression\_case\_OP\_ADDR

至此，我们完成了对一元运算符表达式的语义检查，在后续章节中，我们会对二元表

达式进行语义检查。

#### 4.2.8 二元表达式、赋值表达式和条件表达式的语义检查

在图 4.3 中我们已初步介绍过用于对二元表达式进行语义检查的入口函数 CheckBinaryExpression()。在本小节中，我们准备对图 4.3 第 2 至 19 行中列出的各个具体的函数进行讨论。对于形如  $a+b$  的二元表达式，如果  $a$  为 int 型且  $b$  为 double 型，则表达式  $a+b$  的类型为 double，通过在前面章节中介绍的函数 CommonRealType() 可求得整个表达式  $a+b$  的类型。在图 4.35 中给出的宏定义 PERFORM\_ARITH\_CONVERSION() 中，第 9 行调用了 CommonRealType() 函数用来求公共类型，而第 10 至 11 行对二元运算符的左操作数和右操作数进行必要的转型操作。图 4.35 第 1 行的宏 SWAP\_KIDS 用于交换左右操作数，例如当遇到形如  $i + \text{ptr}$  的表达式，其中  $i$  为整数，而  $\text{ptr}$  为指针，如果希望左操作数为指针类型，而右操作数为整数类型，则可使用宏 SWAP\_KIDS 来交换左右操作数，进而得到  $\text{ptr}+i$ 。第 14 行的 REPORT\_OP\_ERROR 用于报错，表示遇到了非法的运算数。

```

1 #define SWAP_KIDS(expr)           \
2 {                                \
3     AstExpression t = expr->kids[0]; \
4     expr->kids[0] = expr->kids[1]; \
5     expr->kids[1] = t;              \
6 }
7
8 #define PERFORM_ARITH_CONVERSION(expr) \
9     expr->ty = CommonRealType(expr->kids[0]->ty, expr->kids[1]->ty); \
10    expr->kids[0] = Cast(expr->ty, expr->kids[0]);                   \
11    expr->kids[1] = Cast(expr->ty, expr->kids[1]);
12
13
14 #define REPORT_OP_ERROR          \
15     Error(&expr->coord, "Invalid operands to %s", OPNames[expr->op]); \
16     expr->ty = T(INT);           \
17     return expr;

```

图 4.35 宏 PERFORM\_ARITH\_CONVERSION

接下来我们来分析一下在图 4.3 中列出的 CheckEqualityOP() 函数，其代码如图 4.36 所示。第 2 行我们给出了用于判断是否为算术类型的宏 IsArithType，其定义在 ucltype.h，第 3 行的 IsScalarType 用于判断类型是否为标量类型。结合第 3 行的 POINTER 等枚举常量及其在 type.h 中的枚举定义，不难读懂 type.h 中的其他几个宏，如 BothScalarType 等，这里不再啰嗦。图 4.36 第 5 至 24 行的代码用于对形如  $a==b$  或  $a!=b$  的表达式进行语义检查，如果操作数  $a$  和  $b$  都是算术类型（即整型和浮点类型），则在第 12 行用宏 PERFORM\_ARITH\_CONVERSION 来完成必要的类型转换。表达式  $a==b$  或  $a!=b$  的结果为真或假，在 C 语言中，我们用 int 类型来表示布尔值，所以第 13 行置表达式的类型为 int。第 14 行调用 FoldConstant 函数进行必要的常量折叠，例如对于  $3 == 2$  这样的表达式，没有必要到运行时再求值，编译时我们就可以知道表达式  $3 == 2$  的结果为 0。

```

1 // 
2 #define IsArithType(ty)   (ty->categ <= LONGDOUBLE)
3 #define IsScalarType(ty)  (ty->categ <= POINTER)
4 #define IsPtrType(ty)     (ty->categ == POINTER)
5 // a == b ,      a != b
6 static AstExpression CheckEqualityOP(AstExpression expr) {

```

```

7  Type ty1, ty2;
8  expr->ty = T(INT);
9  ty1 = expr->kids[0]->ty;
10 ty2 = expr->kids[1]->ty;
11 if (BothArithType(ty1, ty2)){
12     PERFORM_ARITH_CONVERSION(expr);
13     expr->ty = T(INT);
14     return FoldConstant(expr);
15 }
16 if (IsCompatiblePtr(ty1, ty2) ||
17     NotFunctionPtr(ty1) && IsVoidPtr(ty2) ||
18     NotFunctionPtr(ty2) && IsVoidPtr(ty1) ||
19     IsPtrType(ty1) && IsNullConstant(expr->kids[1]) ||
20     IsPtrType(ty2) && IsNullConstant(expr->kids[0])){
21     return expr;
22 }
23 REPORT_OP_ERROR;
24 }
25 // a&b, a|b, a^b
26 static AstExpression CheckBitwiseOP(AstExpression expr){
27 if (BothIntegType(expr->kids[0]->ty, expr->kids[1]->ty)) {
28     PERFORM_ARITH_CONVERSION(expr);
29     return FoldConstant(expr);
30 }
31 REPORT_OP_ERROR;
32 }
33 static AstExpression CheckLogicalOP(AstExpression expr){//a&&b, a||b
34 if (BothScalarType(expr->kids[0]->ty, expr->kids[1]->ty)) {
35     expr->ty = T(INT);
36     return FoldConstant(expr);
37 }
38 REPORT_OP_ERROR;
39 }
40 // a*b, a/b, a%b
41 static AstExpression CheckMultiplicativeOP(AstExpression expr){
42 if (expr->op != OP_MOD &&
43     BothArithType(expr->kids[0]->ty, expr->kids[1]->ty)) {
44     goto ok;
45 }
46 if (expr->op == OP_MOD &&
47     BothIntegType(expr->kids[0]->ty, expr->kids[1]->ty)) {
48     goto ok;
49 }
50 REPORT_OP_ERROR;
51 ok:
52 PERFORM_ARITH_CONVERSION(expr);
53 return FoldConstant(expr);
54 }
55 // a << b, a >> b
56 static AstExpression CheckShiftOP(AstExpression expr){
57 if (BothIntegType(expr->kids[0]->ty, expr->kids[1]->ty)) {
58     expr->kids[0] = DoIntegerPromotion(expr->kids[0]);
59     expr->kids[1] = DoIntegerPromotion(expr->kids[1]);

```

```

60     expr->ty = expr->kids[0]->ty;
61     return FoldConstant(expr);
62 }
63 REPORT_OP_ERROR;
64 }

```

图 4.36 CheckEqualityOP()

图 4.36 第 16 至 21 行是按照 C 标准 ansi.c.txt 中的语义规则进行编写的，如下所示：

- (1) 两个指针是类型相容的指针，我们在前面的章节中介绍过“相容类型”的概念。对应图 4.36 第 16 行。

(2) 一个指针是指向数据对象的指针，另一个是 void \*。对应图 4.36 第 17 至 18 行。

(3) 一个是指针类型，而另一个为 NULL。对应上图 4.36 第 19 至 20 行。

图 4.36 第 25 至 32 行用于处理 a&b、a|b 和 a^b 这样的位运算，这些运算符要求操作数为整型；而第 33 至 39 行则用于处理形如 a && b 和 a||b 这样的短路运算，第 34 行的 if 条件要求两个操作数都为标量类型（即整型、浮点型和指针类型等）；第 41 至 54 行的代码用于对 a/b、a\*b 和 a%b 进行语义检查，取余运算%要求两个操作数都为整型，而乘除运算则可以是整型或浮点型。第 55 至 61 行的函数 CheckShiftOP() 则用于检查形如 a<<b 和 a>>b 这样的移位运算，这里需要注意的是，我们并没有使用宏 PERFORM\_ARITH\_CONVERSION 来求公共类型，其原因在于表达式 a>>b 是进行算术右移还是逻辑右移，要取决于“左操作数 a 为有符号整数还是无符号整数”，所以在第 60 行我们置表达式 a>>b 的类型为左操作数 a 的类型。图 4.36 中的代码不算复杂，我们就不再啰嗦。

接下来，我们来分析一下 CheckAddOP() 等函数，如图 4.37 所示。第 1 至 26 行的代码用于处理形如 a+b 的表达式，而第 27 至 48 行的代码则对形如 a-b 的表达式进行语义检查。对于 a+b 而言，第 9 至 11 行用于处理两个操作数都为算术类型的情况；当一个操作数为指针类型，而另一个操作数为整型，表达式 ptr+i 进行的是 C 语言的指针加法运算，在汇编代码层次真正执行的加法为 ptr + k，其中 k 为 i\*sizeof(\*ptr)，第 21 行调用 ScalePointerOffset() 函数来构造进行乘法运算的语法树结点。而对于 a-b 来说，第 30 至 33 行用于处理减法的两个操作数都是算术类型的情况，第 34 至 39 行用于处理形如 ptr-i 的指针运算，跟 ptr+i 的指针运算类似，我们需要在第 37 行调用 ScalePointerOffset 函数()来进行 i\*sizeof(\*ptr) 的乘法运算。第 41 至 48 行则用于处理形如 ptr1-ptr2 的指针减法运算，例如，对于 int arr[3]而言，&arr[2]-&arr[0] 的含义是 &arr[2] 与 &arr[0] 之间相差几个 int 整数，而不是它们的地址相差几个字节，因此在汇编代码层次，我们真正执行的运算为 (ptr2-ptr1)/sizeof(\*arr)，第 44 行调用函数 PointerDifference() 来构造相应的除法运算结点。第 49 至 60 行给出了该函数的代码，第 53 行的 CREATE\_AST\_NODE 用于创建语法树结点，第 55 行置其运算符为 OP\_DIV，即除法。

```

1 static AstExpression CheckAddOP(AstExpression expr) {
2     Type ty1, ty2;
3     if (expr->kids[0]->op == OP_CONST) {
4         SWAP_KIDS(expr);
5     }
6     ty1 = expr->kids[0]->ty;
7     ty2 = expr->kids[1]->ty;
8     if (BothArithType(ty1, ty2)){ // 3 + 5
9         PERFORM_ARITH_CONVERSION(expr);
10    return FoldConstant(expr);
11 }
12 if (IsObjectPtr(ty2) && IsIntegType(ty1)){ // i + ptr
13    SWAP_KIDS(expr);

```

```

14     ty1 = expr->kids[0]->ty;
15     goto left_ptr;
16 }
17 if (IsObjectPtr(ty1) && IsIntegType(ty2)){ // ptr + i
18 left_ptr: // ptr + i ---> ptr + i * sizeof(*ptr)
19     expr->kids[1] = DoIntegerPromotion(expr->kids[1]);
20     expr->kids[1] =
21         ScalePointerOffset(expr->kids[1], ty1->bty->size);
22     expr->ty = ty1;
23     return expr;
24 }
25 REPORT_OP_ERROR;
26 }
27 static AstExpression CheckSubOP(AstExpression expr){
28 Type ty1, ty2;
29 ty1 = expr->kids[0]->ty; ty2 = expr->kids[1]->ty;
30 if (BothArithType(ty1, ty2)){ // 5-3
31     PERFORM_ARITH_CONVERSION(expr);
32     return FoldConstant(expr);
33 }
34 if (IsObjectPtr(ty1) && IsIntegType(ty2)){ // ptr - k
35     expr->kids[1] = DoIntegerPromotion(expr->kids[1]);
36     expr->kids[1] =
37         ScalePointerOffset(expr->kids[1], ty1->bty->size);
38     expr->ty = ty1;
39     return expr;
40 }
41 if (IsCompatiblePtr(ty1, ty2)){ // ptr1 - ptr2
42     // ptr1 - ptr2 ---> (ptr1 - ptr2)/sizeof(*ptr2)
43     expr->ty = T(INT);
44     expr = PointerDifference(expr, ty1->bty->size);
45     return expr;
46 }
47 REPORT_OP_ERROR;
48 }
49 static AstExpression PointerDifference(
50             AstExpression diff, int size){
51 AstExpression expr;
52 union value val;
53 CREATE_AST_NODE(expr, Expression);
54 expr->ty = diff->ty;
55 expr->op = OP_DIV;
56 expr->kids[0] = diff;
57 val.i[1] = 0; val.i[0] = size;
58 expr->kids[1] = Constant(diff->coord, diff->ty, val);
59 return expr;
60 }

```

图 4.37 CheckAddOP() 和 CheckSubOP()

下面，我们来讨论赋值运算表达式的语义检查。按照 C 的语义，在表达式  $a = b$  中， $a$  只被计算一次，而在  $a = a + b$  中， $a$  却要被计算两次。例如，对以下代码而言，在表达式  $*f() += 3$  中，函数  $f()$  只被调用一次，而在表达式  $*f() += *f() + 3$  中，函数  $f()$  需要被调用两次。因此，当我们把  $a += b$  转换为  $a = a + b$  来处理，也要保持这样的语义不变。

```

int * f(void) {
    static int number;
    printf("int * f(void) \n");
    return &number;
}
int main(int argc,char * argv[]){
    *f() += 3;
    *f() += *f() + 3;
    return 0;
}

```

对于在 C 源代码中就写为  $a = a + b$  的表达式来说（为表述方便，不妨记为  $a_1 = a_2 + b$ ），在语法树上，有两个结点与  $a$  对应，即  $a_1$  和  $a_2$  各对应一个语法树结点；但对  $a += b$  而言，只有一个语法树结点与  $a$  对应，语义检查时，我们把  $a += b$  转换为  $a = a' + b$  来处理后，并不会为  $a$  构造新的语法树结点。为了与  $a_1 = a_2 + b$  有所区别，在以后的讨论中，不妨把由  $a += b$  得来的表达式写为  $a = a' + b$ ，其中  $a$  和  $a'$  对应的是同一个语法树结点。

有了这个基础后，让我们来分析一下对赋值表达式进行语义检查的函数 CheckAssignmentExpression()。赋值运算符左侧的操作数必须是左值，且该操作数在声明时不应有限定符 const，如果左侧操作数是结构体对象，则在该结构体的定义中，所有的成员域都不应有 const 限定符，图 4.38 第 38 至 44 行的 CanModify() 函数完成了这些判断。第 13 行调用 CanModify() 函数来检查一下左操作数是否为可写的左值。第 18 至 27 行用于把形如  $a += b$  的表达式转换为  $a = a' + b$  来处理，我们只在第 20 行创建了一个新的语法树结点用来存放运算符  $=$ ，而  $a$  和  $a'$  始终对应同一个语法树结点。对于  $a = a' + b$  来说，第 26 行实际上是调用 CheckAddOP() 函数来对加法运算进行语义检查。第 29 行调用的 CanAssign() 函数用于检测赋值运算符两侧的操作数在类型上是否匹配，我们已在前面的章节中分析过 CanAssign() 函数。

```

1 //      a=b , a*=b , /= %= += -= <<= >>=     &= ^= |=
2 static AstExpression CheckAssignmentExpression(
3         AstExpression expr) {
4     int ops[] = {
5         OP_BITOR, OP_BITXOR, OP_BITAND, OP_LSHIFT, OP_RSHIFT,
6         OP_ADD,   OP_SUB,    OP_MUL,    OP_DIV,    OP_MOD
7     };
8     Type ty;
9     expr->kids[0] =
10    Adjust(CheckExpression(expr->kids[0]), 0);
11    expr->kids[1] =
12    Adjust(CheckExpression(expr->kids[1]), 1);
13    if (!CanModify(expr->kids[0])){
14        Error(&expr->coord,
15              "The left operand cannot be modified");
16    }
17 // a += b -----> a = a' + b;
18    if (expr->op != OP_ASSIGN){
19        AstExpression lopr;
20        CREATE_AST_NODE(lopr, Expression);
21        lopr->coord = expr->coord;
22        lopr->op     = ops[expr->op - OP_BITOR_ASSIGN];
23        lopr->kids[0] = expr->kids[0];
24        lopr->kids[1] = expr->kids[1];
25        expr->kids[1] =

```

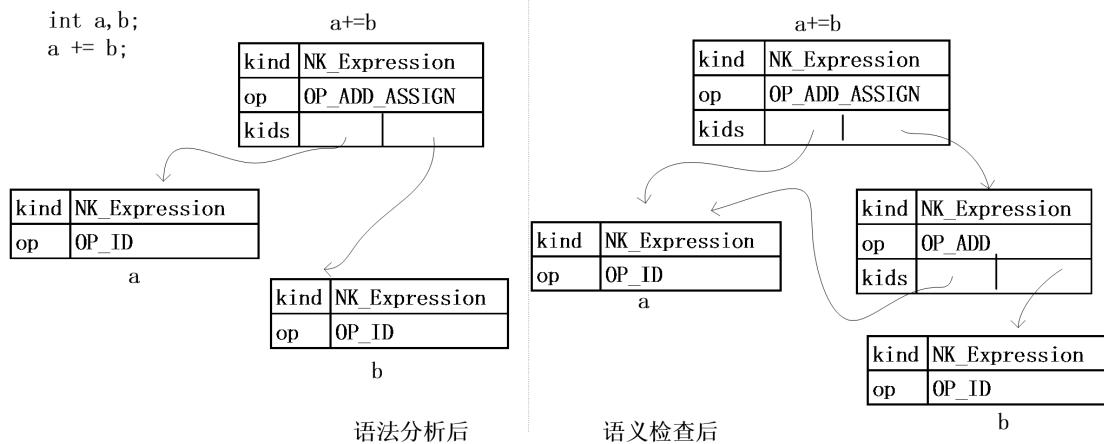
```

26             (*BinaryOPCheckers[lopr->op - OP_OR])(lopr);
27     }
28     ty = expr->kids[0]->ty;
29     if (!CanAssign(ty, expr->kids[1])){
30         Error(&expr->coord, "Wrong assignment");
31     }else{
32         expr->kids[1] = Cast(ty, expr->kids[1]);
33     }
34     expr->ty = ty;
35     return expr;
36 }
37
38 static int CanModify(AstExpression expr){
39     return (
40         expr->lvalue && ! (expr->ty->qual & CONST) &&
41         (IsRecordType(expr->ty) ?
42          ! ((RecordType)expr->ty)->hasConstFld : 1 )
43     );
44 }

```

图 4.38 CheckAssignmentExpression()

图 4.39 给出了表达式  $a+=b$  在语法分析后和语义检查后的语法树，由该图我们可以很清楚地看到，在把  $a+=b$  转换为  $a=a'+b$  后，语法树上  $a$  对应的结点仍旧只有一个。语义检查后，我们还会在语法树上添加各结点的类型信息，此处为简单起见忽略了这些信息。

图 4.39  $a+=b$  的语法树

然后，我们来讨论一下形如  $a?b:c$  的条件表达式的语义检查，与其相关的代码如图 4.40 所示。条件表达式实际上可被当作是三元运算符，即有  $a$ 、 $b$  和  $c$  这 3 个操作数。图 4.40 第 6 至 12 行用于对第一个操作数  $a$  进行语义检查，第 9 行要求第一个操作数  $a$  为标量类型，第 13 至 18 行递归地调用 `CheckExpression()` 函数，对第 2 个操作数  $b$  和第 3 个操作数  $c$  分别进行语义检查。

```

1 // a ? b : c
2 static AstExpression CheckConditionalExpression(
3     AstExpression expr) {
4     int qual;
5     Type ty1, ty2;
6     expr->kids[0] =
7         Adjust(CheckExpression(expr->kids[0]), 1);
8     // The first operand shall have scalar type.
9     if (!IsScalarType(expr->kids[0]->ty)) {

```

```

10     Error(&expr->coord,
11           "The first expression shall be scalar type.");
12 }
13 // the second operand
14 expr->kids[1]->kids[0] =
15     Adjust(CheckExpression(expr->kids[1]->kids[0]), 1);
16 // the third operand
17 expr->kids[1]->kids[1] =
18     Adjust(CheckExpression(expr->kids[1]->kids[1]), 1);
19 ty1 = expr->kids[1]->kids[0]->ty;
20 ty2 = expr->kids[1]->kids[1]->ty;
21 // check types
22 if (BothArithType(ty1, ty2)){
23     expr->ty = CommonRealType(ty1, ty2);
24     expr->kids[1]->kids[0] =
25         Cast(expr->ty, expr->kids[1]->kids[0]);
26     expr->kids[1]->kids[1] =
27         Cast(expr->ty, expr->kids[1]->kids[1]);
28     return FoldConstant(expr);
29 }else if (IsRecordType(ty1) && ty1 == ty2){
30     expr->ty = ty1;
31 } else if (ty1->categ == VOID && ty2->categ == VOID) {
32     expr->ty = T(VOID);
33 }else if (IsCompatiblePtr(ty1, ty2)){
34     qual = ty1->bty->qual | ty2->bty->qual;
35     expr->ty = PointerTo(Qualify(qual,
36         CompositeType(Unqual(ty1->bty), Unqual(ty2->bty))));
37 } else if (IsPtrType(ty1) &&
38             IsNullConstant(expr->kids[1]->kids[1])){
39     expr->ty = ty1;
40 } else if (IsPtrType(ty2) &&
41             IsNullConstant(expr->kids[1]->kids[0])){
42     expr->ty = ty2;
43 }else if (NotFunctionPtr(ty1) && IsVoidPtr(ty2) ||
44             NotFunctionPtr(ty2) && IsVoidPtr(ty1)){
45     qual = ty1->bty->qual | ty2->bty->qual;
46     expr->ty = PointerTo(Qualify(qual, T(VOID)));
47 } else{
48     Error(&expr->coord, "invalid operand for ? operator.");
49     expr->ty = T(INT);
50 }
51 return expr;
52 }

```

图 4.40 CheckConditionalExpression()

图 4.40 第 19 至 50 行的代码，用于对形如  $a?b:c$  中的操作数  $b$  和  $c$  的类型进行检查，这些代码是按照 C 标准 ansi.c.txt 的第 3.3.15 节的语义规则编写的，如下所示：

- (1)  $b$  和  $c$  的类型都为算术类型，对应图 4.40 第 22 至 28 行，此时整个条件表达式的类型为  $b$  和  $c$  的公共类型。
- (2)  $b$  和  $c$  的类型为相容的结构体或联合体类型，对应图 4.40 第 29 至 30 行。
- (3)  $b$  和  $c$  的类型都为 void，对应图 4.40 第 31 至 32 行，此时整个条件表达式的类型就为 void。

(4) b 和 c 为相容的指针类型, 对应图 4.40 第 33 至 36 行, 此时整个条件表达式的类型为 b 和 c 类型的合成类型。

(5) 一个为指针类型, 另一个为 NULL, 对应图 4.40 第 37 至 42 行。

(6) 一个是指向数据对象的指针 (即不是指向函数的指针), 另一个是 void \*, 对应图 4.40 第 43 至 46 行。

除了以上这 6 种情况外, b 和 c 的其他类型都被视为非法的, 图 4.40 第 48 行会进行报错。图 4.40 中调用的 CommonRealType() 和 CompositeType() 等函数, 我们都已在前面的章节中做过讨论。至此, 我们完成了对 C 语言表达式的语义检查, 相关的代码主要在 exprchk.c 中, 可以发现后缀表达式和一元表达式的语义是相对比较复杂的, 例如, 后缀运算符[]和一元运算符\*都涉及到了内存寻址。对这些运算符的语义检查, 有助于我们更深入地理解 C 语言。

### 4.3 语句的语义检查

在这一节中, 我们来分析一下语句的语义检查, 与其相关的代码在 stmtchk.c 中。图 4.41 第 1 至 16 行的数组 StmtCheckers, 给出了用于对各语句进行语义检查的函数表, 第 17 行的 CheckStatement(stmt) 函数根据参数 stmt 的 kind 域来检索这个函数表, 从而调用相应的函数进行语义检查, 如图第 18 行所示。

```

1 static AstStatement (* StmtCheckers[]) (AstStatement) = {
2     CheckExpressionStatement,
3     CheckLabelStatement,
4     CheckCaseStatement,
5     CheckDefaultStatement,
6     CheckIfStatement,
7     CheckSwitchStatement,
8     CheckLoopStatement,
9     CheckLoopStatement,
10    CheckForStatement,
11    CheckGotoStatement,
12    CheckBreakStatement,
13    CheckContinueStatement,
14    CheckReturnStatement,
15    CheckLocalCompound
16 };
17 static AstStatement CheckStatement(AstStatement stmt) {
18     return (* StmtCheckers[stmt->kind - NK_ExpressionStatement])(stmt);
19 }
```

图 4.41 CheckStatement()

在完成对表达式语义检查的基础上, 再对 switch 和 while 等语句进行语义检查就相对容易了许多。图 4.41 第 15 行的 CheckLocalCompound 用于对复合语句进行语义检查, 其代码如图 4.42 第 20 至 26 行所示。复合语句的左大括号表示了一个新的作用域的开始, 我们需要创建新的符号表来存放在该作用域中声明的符号, 第 22 行调用的 EnterScope 函数中完成了创建新符号表的工作。在图 4.42 第 30 行, 我们把“代表当前作用域深度”的全局变量 Level 加 1, 第 31 至 35 行于创建一个新的符号表, 由第 33 行的 outer 指向外层 (深度较浅的) 符号表, 第 35 行的全局变量 Identifiers 始终指向与当前作用域对应的标志符符号表 (存放变量名和函数名等); 第 37 至 42 行用于创建一个新的用于存放结构体名、枚举名和联合体名的符号表, 第 41 行的全局变量 Tags 指向这个新的符号表。当离开一个作用域时, 即遇到

复合语句的右大括号时，我们就在第 24 行调用 ExitScope() 函数，从而使 Identifiers 和 Tags 指针指向外层的符号表，同时使 Level 渏 1，如图第 44 至 48 行所示。第 23 行调用 CheckCompoundStatement() 函数，对复合语句中出现的若干个声明和若干个语句进行语义检查。

```

1 // the body of function definition
2 AstStatement CheckCompoundStatement(AstStatement stmt) {
3     AstCompoundStatement compStmt = AsComp(stmt);
4     AstNode p;
5
6     compStmt->ilocals = CreateVector(1);
7     p = compStmt->decls;
8     while (p) {
9         CheckLocalDeclaration((AstDeclaration)p, compStmt->ilocals);
10        p = p->next;
11    }
12    p = compStmt->stmts;
13    while (p) {
14        CheckStatement((AstStatement)p);
15        p = p->next;
16    }
17    return stmt;
18 }
19 // local compound statement in function body
20 static AstStatement CheckLocalCompound(AstStatement stmt) {
21     AstStatement s;
22     EnterScope();
23     s = CheckCompoundStatement(stmt);
24     ExitScope();
25     return stmt;
26 }
27 void EnterScope(void) {
28     Table t;
29
30     Level++;
31     ALLOC(t);
32     t->level = Level;
33     t->outer = Identifiers;
34     t->buckets = NULL;
35     Identifiers = t;
36
37     ALLOC(t);
38     t->level = Level;
39     t->outer = Tags;
40     t->buckets = NULL;
41     Tags = t;
42 }
43
44 void ExitScope(void) {
45     Level--;
46     Identifiers = Identifiers->outer;
47     Tags = Tags->outer;

```

```
48 }
```

图 4.42 CheckCompoundStatement()

图 4.42 第 6 行创建了一个向量对象，用于存放在复合语句中声明的局部变量，第 7 至 11 行调用函数 CheckLocalDeclaration() 来对复合语句中的局部声明进行语义检查，我们会在讨论 declchk.c 时分析这个函数。第 12 至 16 行则递归地调用 CheckStatement() 函数，来对复合语句中的各个语句进行语义检查。对于以下代码来说，形参 a、形参 b 和局部变量 c 实际上处于同一个作用域，而局部变量 d 处于更深的作用域。其中，标注为 Level1 的复合语句是函数 f 的函数体，我们在对形参 a 和 b 进行语义检查时，就要调用 EnterScope() 来创建新的符号表，而遇到标注为 Level1 的复合语句时就不需要再调用 EnterScope()。图 4.42 第 2 行的 CheckCompoundStatement() 函数，用于对如下标为 Level1 的复合语句进行语义检查，在 CheckCompoundStatement() 中并没有调用 EnterScope()。而图 4.42 第 20 行的 CheckLocalCompound() 则用于对函数体中的复合语句进行语义检查，例如以下标注为 Level2 的复合语句，我们需要调用 EnterScope() 来创建新的符号表。

```
void f(int a,int b){ // Level1
    int c;
    { // Level2
        int d;
    }
}
```

在 C 语言中，break 语句必须出现在 switch 语句、for 语句、while 语句或者 do\_while 语句中，在 UCC 编译器中，这些语句被称为 breakable。UCC 编译器的全局变量 CURRENTF 指向当前函数对应的语法树，向量 CURRENTF->breakable 用于存放这些 breakable 语句。而 continue 语句则要出现在 for 语句、while 语句或者 do\_while 语句中，这些被称为循环语句，向量 CURRENTF->loops 用于存放这些循环语句。而 case 和 default 语句必须出现在 switch 语句中，向量 CURRENTF->swtches 用于存放 switch 语句

我们可以把 CURRENTF->swtches 向量看成一个栈，在语义检查时，当遇到 switch 语句时，我们把 switch 语句入栈，在完成对 switch 语句的语义检查时把 switch 语句从 CURRENTF->swtches 中出栈。如果遇到 case 语句，则检查一下栈 CURRENTF->swtches 是否为空，如果不空，说明该 case 语句确实是出现在栈顶 switch 语句中；否则该 cast 语句出现的位置非法。按与此类似的思路，我们也可以对 break 和 continue 语句进行语义检查。图 4.43 第 1 行的宏 PushStatement(v,stmt) 用于把语句 stmt 入栈 v，而第 2 行的宏 PopStatement(v) 则用于把栈 v 的栈顶元素出栈，第 3 行的 TopStatement(v) 可在不出栈的情况下取栈顶元素。

```
1 #define PushStatement (v, stmt)    INSERT_ITEM(v, stmt)
2 #define PopStatement (v)          (v->data[--v->len])
3 #define TopStatement (v)          TOP_ITEM(v)
4 static AstStatement CheckSwitchStatement (AstStatement stmt) {
5     AstSwitchStatement swtchStmt = AsSwitch(stmt);
6     PushStatement (CURRENTF->swtches, stmt);
7     PushStatement (CURRENTF->breakable, stmt);
8     swtchStmt->expr =
9         Adjust(CheckExpression(swtchStmt->expr), 1);
10    if (! IsIntegType(swtchStmt->expr->ty)) {
11        Error(&stmt->coord,
12              "The expression in a switch shall be integer type.");
13        swtchStmt->expr->ty = T(INT);
14    }
15    if (swtchStmt->expr->ty->categ < INT) {
16        swtchStmt->expr = Cast(T(INT), swtchStmt->expr);
```

```

17 }
18 swtchStmt->stmt = CheckStatement(swtchStmt->stmt);
19 PopStatement(CURRENTF->swtches);
20 PopStatement(CURRENTF->breakable);
21 return stmt;
22 }
23 static AstStatement CheckCaseStatement(AstStatement stmt) {
24 AstCaseStatement caseStmt = AsCase(stmt);
25 AstSwitchStatement swtchStmt;
26 swtchStmt =
27     (AstSwitchStatement)TopStatement(CURRENTF->swtches);
28 if (swtchStmt == NULL){
29     Error(&stmt->coord,
30         "A case label shall appear in a switch statement.");
31     return stmt;
32 }
33 caseStmt->expr = CheckConstantExpression(caseStmt->expr);
34 if (caseStmt->expr == NULL){
35     Error(&stmt->coord,
36         "The case value must be integer constant.");
37     return stmt;
38 }
39 caseStmt->stmt = CheckStatement(caseStmt->stmt);
40 caseStmt->expr =
41     FoldCast(swtchStmt->expr->ty, caseStmt->expr);
42 AddCase(swtchStmt, caseStmt);
43 return stmt;
44 }
45 static AstStatement CheckDefaultStatement(AstStatement stmt) {
46 AstDefaultStatement defStmt = AsDef(stmt);
47 AstSwitchStatement swtchStmt;
48 swtchStmt = (AstSwitchStatement)TopStatement(CURRENTF->swtches);
49 if (swtchStmt == NULL){
50     Error(&stmt->coord, "A default label shall appear in a switch statement.");
51     return stmt;
52 }
53 if (swtchStmt->defStmt != NULL){
54     Error(&stmt->coord,
55         "There shall be only one default label in a switch statement.");
56     return stmt;
57 }
58 defStmt->stmt = CheckStatement(defStmt->stmt);
59 swtchStmt->defStmt = defStmt;
60 return stmt;
61 }

```

图 4.43 CheckSwitchStatement()

图 4.43 第 4 至 22 行的代码用于对 switch 语句进行语义检查, 第 6 和第 7 行把当前 switch 语句分别入栈 CURRENTF->swtches 和 CURRENTF->breakable。第 8 至 17 行用于对 switch(expr) 中的表达式 expr 进行语义检查, 我们已在 4.2 节讨论过第 9 行调用的 CheckExpression() 函数。第 18 行递归地调用 CheckStatement() 函数对 switch 中的语句 swtchStmt->stmt 进行语义检查, 此时如果遇到 case 语句, 则栈 CURRENTF->swtches 不为空。第 19 至 20 行把当前 switch 语句分别从 CURRENTF->swtches 和 CURRENTF->breakable

中出栈，表示我们已经完成了对当前 switch 语句的语义检查。

图 4.43 第 23 至 44 行的代码用于检查 case 语句，与其 case 语句相关的产生式如下所示。图 4.43 第 26 至 32 行取 CURRENTF->swtches 的栈顶元素，如果为空，则说明该 case 语句没有出现在 switch 语句中，我们需要在第 29 行进行报错。第 33 至 38 行则调用函数 CheckConstantExpression() 来检查一下，case 语句中出现的表达式是否为编译时的常量，即产生式中的 constant-expression，第 39 行则递归地调用 CheckStatement() 来对产生式中的 statement 进行语义检查。第 42 行把整个 case 语句加入到相应的 switch 语句中，以便于后续的中间代码生成。

```
case-statement:
    case constant-expression : statement
```

图 4.43 第 45 至 60 行的函数 CheckDefaultStatement 用于对 default 语句进行语义检查，第 48 至 52 行检查一下 CURRENTF->swtches 的栈顶是否为空。由于一个 switch 语句中，只能有一个 default 语句，我们需要在第 53 至 57 行检查一下之前是否已有 default 语句。

而 stmtchk.c 中的其他函数，例如 CheckLoopStatement() 和 CheckForStatement() 等，基本上与图 4.43 类似，在理解图 4.43 的基础上，不难理解这些函数。我们就不再啰嗦。在下一节中，我们准备对 declchk.c 中的代码进行讨论，即对声明进行语义检查，从而建立 C 语言的类型系统。

## 4.4 声明的语义检查

### 4.4.1 类型结构的构建

在这一节中，我们要对 C 语言的声明进行语义检查，主要有“外部声明”和“出现在复合语句中的局部声明”，相关代码主要在 declchk.c 中。按 C 标准文法，“函数定义”和“在函数体外出现的变量与函数声明”都是外部声明。每个 C 文件在文法中被称为翻译单元，与之相关的语义检查函数如图 4.44 第 1 至第 15 行所示。第 7 行调用的 CheckFunction() 用于对函数定义进行语义检查，而第 10 行调用的 CheckGlobalDeclaration() 用于对“在函数体外出现的变量和函数声明”进行语义检查。第 16 至 33 行是函数 CheckGlobalDeclaration() 的核心代码，第 34 至 49 行则为 CheckFunction() 的核心代码。对比这两者的代码，我们可以发现它们都依次调用了对“声明说明符”和“声明符”进行语义检查的函数。而被声明的标识符的类型信息分布在“声明说明符”和“声明符”中，通过调用第 29 或 46 行的 DeriveType() 函数，会把这两部分的类型信息组合到一起，从而创建完整的类型结构。例如，对于 int arr[4] 来说，int 是其声明说明符，而 arr[4] 是声明符，只有把 int 和 [4] 组合到一起，我们才能得到 arr 的类型信息为 int [4]。

```
1 void CheckTranslationUnit(AstTranslationUnit transUnit) {
2     AstNode p;
3     Symbol f;
4     p = transUnit->extDecls;
5     while (p) {
6         if (p->kind == NK_Function) {
7             CheckFunction((AstFunction)p);
8         } else{
9             assert(p->kind == NK_Declaration);
10            CheckGlobalDeclaration((AstDeclaration)p);
11        }
12        p = p->next;
```

```

13 }
14 ...
15 }
16 static void CheckGlobalDeclaration(AstDeclaration decl) {
17   AstInitDeclarator initDec;
18   Type ty;
19   Symbol sym;
20   int sclass;
21   CheckDeclarationSpecifiers(decl->specs);
22   .....
23   initDec = (AstInitDeclarator) decl->initDecls;
24   while (initDec) {
25     CheckDeclarator(initDec->dec);
26     // int ; -----> anonymous
27     if (initDec->dec->id == NULL)
28       goto next;
29     ty = DeriveType(initDec->dec->tyDrvList,
30                      decl->specs->ty, &initDec->coord);
31     .....
32   }
33 }
34 void CheckFunction(AstFunction func) {
35   Symbol sym;
36   Type ty;
37   int sclass;
38   Label label;
39   AstNode p;
40   .....
41   CheckDeclarationSpecifiers(func->specs);
42   .....
43   // check declarator
44   CheckDeclarator(func->dec);
45   .....
46   ty = DeriveType(func->dec->tyDrvList,
47                     func->specs->ty, &func->coord);
48   .....
49 }

```

图 4.44 CheckTranslationUnit()

我们在第3章图3.36给出了“声明”对应的语法树，在图3.39给出了“函数定义”对应的语法树。如果把函数定义看成是“函数声明+函数体（即复合语句）”，则函数定义的类型信息主要体现在其函数声明部分。

构建类型结构的工作主要由图4.44第21行调用的函数CheckDeclarationSpecifiers()，第25行调用的CheckDeclarator()函数和第29行调用的DeriveType()函数来完成。再次强调，阅读相关代码时，一定要结合第3章的语法树和第2.4节的类型结构来进行。

接下来，让我们先分析一下函数CheckDeclarationSpecifiers()。当提到声明说明符时，我们应条件反射般地想到“static const int”，其中static被称为存储类说明符，而const被称为类型限定符，int被称为类型说明符。函数CheckDeclarationSpecifiers()的主要代码如图4.45所示。

```

1 static void CheckDeclarationSpecifiers(AstSpecifiers specs) {
2   AstToken tok;
3   AstNode p;

```

```

4   Type ty;
5   int size = 0, sign = 0;
6   int signCnt = 0, sizeCnt = 0, tyCnt = 0;
7   int qual = 0;
8   //storage-class-specifier: extern,auto,static, register
9   tok = (AstToken)specs->stgClasses;
10  if (tok) {
11      if (tok->next){
12          Error(&specs->coord, "At most one storage class");
13      }
14      specs->sclass = tok->token;
15  }
16 //type-qualifier: const, volatile
17 tok = (AstToken)specs->tyQuals;
18 while (tok){
19     qual |= (tok->token == TK_CONST ? CONST : VOLATILE);
20     tok = (AstToken)tok->next;
21 }
22 //type-specifier: int,double, struct ..., union
23 p = specs->tySpecs;
24 while (p){
25     if (p->kind == NK_StructSpecifier
26         || p->kind == NK_UnionSpecifier){// struct/union
27         ty = CheckStructOrUnionSpecifier((AstStructSpecifier)p);
28         tyCnt++;
29     }else if (p->kind == NK_EnumSpecifier){// enum
30         ty = CheckEnumSpecifier((AstEnumSpecifier)p);
31         tyCnt++;
32     }else if (p->kind == NK_TypedefName){ // INT32
33         Symbol sym = LookupID(((AstTypedefName)p)->id);
34         ty = sym->ty;
35         tyCnt++;
36     }else{ // int, char , float, double, ...
37         tok = (AstToken)p;
38         switch (tok->token) {
39             .....
40             case TK_INT:
41                 ty = T(INT);
42                 tyCnt++;
43                 break;
44             .....
45         }
46     }
47     p = p->next;
48 }
49 .....
50 specs->ty = Qualify(qual, ty);
51 return;
52 err:
53 Error(&specs->coord, "Illegal type specifier.");
54 specs->ty = T(INT);
55 return;

```

```
56 }
```

图 4.45 CheckDeclarationSpecifiers()

图 4.45 第 8 至 15 行用来检查存储类说明符，按 C 标准，一个声明中最多只能有一个存储类说明符，即只能是 `extern`、`auto`、`static` 或 `register` 中的一个。第 16 至 21 行用来检查类型限定符 `const` 或 `volatile`。第 22 至 48 行则用于对 `int`、`double`、`struct` 和 `enum` 等类型说明符进行检查，我们只给出了主要的代码，省略了一些细节。图 4.45 第 27 行调用函数 `CheckStructOrUnionSpecifier()`，在语法树的基础上为 `struct` 或 `union` 构建类型结构；第 30 行则调用 `CheckEnumSpecifier()` 函数来为 `enum` 构建类型结构；第 32 行用于处理用户通过“`typedef int INT32`”定义的类型名 `INT32`，第 33 行查符号表，从符号表中取出类型名 `INT32` 对应的类型信息，对“`typedef int INT32`”的语义检查会由函数 `CheckTypedef()` 完成；第 36 至 46 行为 `int` 和 `double` 等基本类型构造类型结构，我们只在第 40 行用 `int` 作为示范。第 50 行调用 `Qualify()` 函数，把从“类型说明符”中得到的类型信息 `ty` 与“类型限定符”结合，把由此得到的类型结构存到 `specs->ty` 中。函数 `CheckStructOrUnionSpecifier()` 比较复杂，我们会在稍后进行讨论。在分析完 `CheckStructOrUnionSpecifier()` 后，再去理解函数 `CheckEnumSpecifier()` 和 `CheckTypedef()` 函数，就会容易许多。

在声明中，另一部分的类型信息由“声明符”来指定。在第 3 章图 3.36 和图 3.39 的语法树中，对于 `a2`、`a3` 和 `fn` 来说，经语法分析后，实际上存在这 3 条链表，为讨论方便，我们不妨称这些链表为“Declarator 链表”。

```
// static const int a1 = 3, * a2, a3[5],a4(void);
// int * fn(int aa,double bb,double cc) { return NULL;}
astPointerDeclarator(对应*) ---> astDeclarator(对应 a2);
astArrayDeclarator(对应[]) ---> astDeclarator(对应 a3);
astPointerDeclarator(对应*) ---> astFunctionDeclarator(对应())---> astDeclarator(对应 fn);
```

而函数 `CheckDeclarator()` 要做的事情就是遍历如上所示的某条链表，收集 Declarator 链表上各结点的类型信息，最终再通过 `DeriveType()` 函数，综合由 `CheckDeclarationSpecifiers()` 和 `CheckDeclarator()` 这两个函数得来的类型信息，为标识符 `a2`、`a3` 和 `fn` 构造出完整的类型结构。为了描述图 3.36 和图 3.39 的 `astPointerDeclarator` 结点、`astArrayDeclarator` 结点和 `astFunctionDeclarator` 结点中的类型信息，图 4.46 第 18 至 26 行中引入了结构体 `struct typeDerivList`。图 4.46 第 19 行的 `ctor` 用于描述类型构造符(即指针、数组和函数)，第 21 行的 `len` 用于记录数组的长度(例如 `int arr[3]` 中的 3)，第 22 行的 `qual` 用于记录指针的限定符(例如“`int * const ptr`”中的 `const`)，第 23 行的 `sig` 用于记录函数形参列表的类型。

```
1 static void CheckDeclarator(AstDeclarator dec) {
2     switch (dec->kind) {
3         case NK_NameDeclarator:
4             break;
5         case NK_ArrayDeclarator:
6             CheckArrayDeclarator((AstArrayDeclarator)dec);
7             break;
8         case NK_FunctionDeclarator:
9             CheckFunctionDeclarator((AstFunctionDeclarator)dec);
10            break;
11        case NK_PointerDeclarator:
12            CheckPointerDeclarator((AstPointerDeclarator)dec);
13            break;
14        default:
15            assert(0);
16    }
```

```

17 }
18 typedef struct typeDerivList{
19     int ctor;          // POINTER_TO, ARRAY_OF, FUNCTION_RETURN
20     union {
21         int len;      // array size
22         int qual;    // pointer qualifier
23         Signature sig; // function signature
24     };
25     struct typeDerivList *next;
26 } *TypeDerivList;
27 static void CheckArrayDeclarator(
28     AstArrayDeclarator arrDec) {
29     CheckDeclarator(arrDec->dec);
30     if (arrDec->expr) {
31         if ((arrDec->expr =
32             CheckConstantExpression(arrDec->expr)) == NULL) {
33             Error(&arrDec->coord,
34                 "The size of the array must be integer constant.");
35         }
36     }
37     ALLOC(arrDec->tyDrvList);
38     arrDec->tyDrvList->ctor = ARRAY_OF;
39     arrDec->tyDrvList->len =
40         arrDec->expr ? arrDec->expr->val.i[0] : 0;
41     arrDec->tyDrvList->next = arrDec->dec->tyDrvList;
42     arrDec->id = arrDec->dec->id;
43 }
44 static void CheckPointerDeclarator(
45     AstPointerDeclarator ptrDec) {
46     int qual = 0;
47     AstToken tok = (AstToken)ptrDec->tyQuals;
48     CheckDeclarator(ptrDec->dec);
49     while (tok){ // * const volatile
50         qual |= tok->token == TK_CONST ? CONST : VOLATILE;
51         tok = (AstToken)tok->next;
52     }
53     ALLOC(ptrDec->tyDrvList);
54     ptrDec->tyDrvList->ctor = POINTER_TO;
55     ptrDec->tyDrvList->qual = qual;
56     ptrDec->tyDrvList->next = ptrDec->dec->tyDrvList;
57     ptrDec->id = ptrDec->dec->id;
58 }

```

图 4.46 CheckDeclarator()

函数 CheckDeclarator() 的代码如图 4.46 第 1 至 17 行所示，第 5 至 7 行用于处理当前结点为数组的情况，第 8 至 10 行处理当前结点为函数的情况，第 11 至 13 行处理当前结点为指针的情况，而第 3 至 4 行则对应如上所示的标识符结点 a2、a3 和 fn。第 27 至 43 行的函数 CheckArrayDeclarator() 完成了对数组结点的类型信息收集，其结果存于第 37 行创建的 struct typeDerivList 对象中，第 38 行记录类型构造符为 ARRAY\_OF（即数组），第 39 行记录数组的长度，而第 41 行则指向下一个 declarator 结点的类型信息，最终构成一个由若干个 struct typeDerivList 对象构成的链表。第 42 行则在当前数组结点 astArrayDeclarator 中记录标识符的名称。第 29 行递归地调用 CheckDeclarator() 完成了对 Declarator 链表的逆序遍历。由

于 C 语言在声明数组时，要求数组大小为常数，第 30 至 36 行完成了这个检查。

图 4.46 第 44 行的函数 CheckPointerDeclarator() 用于收集 Declarator 链表中的指针结点所包含的类型信息。例如，对于 “int \* const ptr;” 来说，除了指针\*外，我们还要把类型限定符 const 也包含进来，第 49 至 52 完成了对类型限定符的处理，第 53 至 57 行则创建 struct typeDerivList 对象用来存放类型构造符 POINTER\_TO 和类型限定符，并指向由 Declarator 链表中下一个结点收集来的类型信息。由于在 C 文法的声明说明符中，并未对类型限定符 const 和类型说明符 int 的顺序进行限制，即 const int 和 int const 是等效的，因此，以下 ptr1 和 ptr2 的类型是一样的。类型限定符 const 是修饰指针本身还是修饰其所指向的对象，要看 const 是位于运算符\*的左侧还是右侧。

```
const int * ptr1; // ptr1 指向一个 const int, 但 ptr1 不是 const
int const * ptr2; // ptr2 指向一个 const int, 但 ptr2 不是 const
const int * const ptr3; // ptr3 指向一个 const int, 且 ptr3 为 const
int * const ptr4; // ptr4 指向一个 int, 且 ptr4 为 const
```

而图 4.46 第 9 行调用的 CheckFunctionDeclarator() 函数，则从 astFunctionDeclarator 结点中收集函数形参列表的类型信息。与之相关的代码，如图 4.47 所示。图 4.47 第 7 行创建了一个 struct signature 对象用于保存函数的形参列表的类型信息，第 11 行创建了一个向量对象用于保存各形参，最终我们为函数构造的类型结构可参见第 2 章的图 2.22。对于新式风格的函数，我们会在图 4.47 第 13 行通过调用 CheckParameterTypeList() 获得其形参类型信息。对于第 15 行注释中所示的旧式风格函数定义 f，此处我们仅处理(a,b,c)这部分，而对于其后的 “int a,b; double c;” 我们会在分析 CheckFunction() 函数时再处理，因此在第 18 行我们只是把各个标识符（例如第 15 行的 a,b,c），通过第 18 行的 AddParameter() 函数加入到向量 funcDec->sig->params 中，我们在第 19 行中先把参数的类型信息设为 NULL。按 C 标准的规定，形如 “void f(a,b,c);” 的旧式声明是非法的，我们会在第 21 至 25 行对此进行检查。我们在第 26 行创建了一个 struct typeDerivList 对象，用来保存从 astFunctionDeclarator 结点中收集来的类型信息，第 27 行设置类型构造符为 FUNCTION\_RETURN（即函数类型），第 28 行用于保存形参的类型信息。

```
1 static void CheckFunctionDeclarator(
2     AstFunctionDeclarator dec){
3     AstFunctionDeclarator funcDec =
4         (AstFunctionDeclarator)dec;
5     CheckDeclarator(funcDec->dec);
6     EnterParameterList();
7     ALLOC(funcDec->sig);
8     funcDec->sig->hasProto =
9         funcDec->paramTyList != NULL;
10    funcDec->sig->hasEllipsis = 0;
11    funcDec->sig->params = CreateVector(4);
12    if (funcDec->sig->hasProto){
13        CheckParameterTypeList(funcDec);
14    }else if (funcDec->partOfDef){
15        // int f(a,b,c) int a,b;double c;{ ... }
16        char *id;
17        FOR_EACH_ITEM(char*, id, funcDec->ids)
18            AddParameter(funcDec->sig->params,
19                         id, NULL, 0, &funcDec->coord);
20    ENDFOR
21    }else if (LEN(funcDec->ids)){
22        // void f(a,b,c);
23        Error(&funcDec->coord,
```

```

24         "Identifier list should be in definition.");
25     }
26     ALLOC(funcDec->tyDrvList);
27     funcDec->tyDrvList->ctor = FUNCTION_RETURN;
28     funcDec->tyDrvList->sig = funcDec->sig;
29     funcDec->tyDrvList->next = funcDec->dec->tyDrvList;
30     funcDec->id = funcDec->dec->id;
31     if(funcDec->partOfDef) { //当前处理的函数声明是函数定义的一部分
32         SaveParameterListTable();
33     }
34     LeaveParemeterList();
35 }
36 static int inParameterList = 0;
37 static Table savedIdentifiers, savedTags;
38 int IsInParameterList(void){
39     return inParameterList;
40 }
41 void EnterParameterList(void){
42     inParameterList = 1;
43     EnterScope();
44 }
45 void LeaveParemeterList(void){
46     inParameterList = 0;
47     ExitScope();
48 }
49 void SaveParameterListTable(void){
50     savedIdentifiers = Identifiers;
51     savedTags = Tags;
52 }
53 void RestoreParameterListTable(void){
54     Level++;
55     savedIdentifiers->outer = Identifiers;
56     savedIdentifiers->level = Level;
57     Identifiers = savedIdentifiers;
58     savedTags->outer = Tags;
59     savedTags->level = Level;
60     Tags = savedTags;
61 }

```

图 4.47 CheckFunctionDeclarator()

对于以下代码而言，对 `f(int a,int b)` 和 `g(int c, int d)` 形参列表的类型收集工作都是由函数 `CheckFunctionDeclarator()` 完成。处理形参列表时，我们实际上就进入了一个新的作用域，要调用前文介绍过的 `EnterScope()` 函数来创建新的符号表，但对于函数定义 `g` 而言，其局部变量 `e` 和形参 `d` 位于同一个作用域，相关符号信息存放在相同的符号表中。而对于如下所示的函数 `h` 而言，其返回值是一个指向 `void (void)` 函数的指针，此时 `Declarator` 链表上就会出现两个 `astFunctionDeclarator` 结点，分别对应 `(int a1,int a2)` 和 `(void)`，其中 `(int a1,int a2)` 处于一个作用域，而 `(void)` 又位于另一个不同的作用域。由于 `h` 函数体中的 `a3` 和形参 `a2` 又处在同一个作用域。因此，当我们调用 `CheckFunctionDeclarator()` 函数分析完 `(int a1,int a2)` 时，可先保存当前作用域的符号表，之后我们会在 `CheckFunction()` 中检查 `h` 的函数体时，再恢复先前保存的符号表，这样就可以使 `a1`、`a2` 和 `a3` 处在相同的符号表中。

```

void f(int a, int b);
void g(int c, int d) {

```

```

    int e;
}
void (*h(int a1,int a2))(void) {
    int a3;
    return NULL;
}

```

图 4.47 第 49 至 52 行的函数 SaveParameterListTable()实现了保存当前符号表的功能，而第 53 至 60 行的函数 RestoreParameterListTable()则用于恢复符号表。当我们开始处理形参列表时，我们会在图 4.47 第 6 行调用 EnterParameterList()函数，为新的作用域创建相应的符号表，实际的工作由第 43 行的 EnterScope()完成；当我们检查完形参列表时，在第 34 行调用 LeaveParameterList()函数，实际的工作由第 47 行的 ExitScope 函数完成。对于在形参列表中出现的结构体定义，由于其在外围作用域中是不可见的，所以 GCC、Clang 和 UCC 编译器会给出相应警告，例如以下函数 k 中的 struct A。第 38 行的 IsInParameterList()函数用来判断我们是否处在形参列表中，以便进行报警。只有处理函数定义中的形参列表时，我们才会调用 SaveParameterListTable()函数来保存其符号表，以便在检查函数体时进行恢复，如图 4.47 第 31 至 33 行所示。

```
void k(struct A{int a;} b);
```

在图 4.47 第 13 行中，我们调用了 CheckParameterTypeList()函数来获取形如(int a,int b)的参数列表中的类型信息，相关代码如图 4.48 第 1 至 13 行所示。结合图 3.39 的语法树，我们可以发现第 8 至 11 行的 while 循环实现了对各个形参的检查，第 9 行调用的函数 CheckParameterDeclaration()完成了对某个形参的检查。对于形如 int a 的形式参数，其类型信息也是分散在声明说明符和声明符中，因此其类型信息收集工作也是由第 19、20 和 27 行调用的 3 个函数来完成。在第 21 至 26 行的注释中，我们列出了一些错误的形参声明，为节约篇幅，与此相关的代码我们没有给出。

```

1 // "int a, int b , int c "in f(int a, int b, int c).
2 static void CheckParameterTypeList(
3     AstFunctionDeclarator funcDec){
4     AstParameterTypeList paramTyList = funcDec->paramTyList;
5     AstParameterDeclaration paramDecl;
6     paramDecl =
7         (AstParameterDeclaration)paramTyList->paramDecls;
8     while (paramDecl){
9         CheckParameterDeclaration(funcDec, paramDecl);
10        paramDecl = (AstParameterDeclaration)paramDecl->next;
11    }
12    funcDec->sig->hasEllipsis = paramTyList->ellipsis;
13 }
14 static void CheckParameterDeclaration(
15     AstFunctionDeclarator funcDec,
16     AstParameterDeclaration paramDecl){
17     char *id = NULL;
18     Type ty = NULL;
19     CheckDeclarationSpecifiers(paramDecl->specs);
20     CheckDeclarator(paramDecl->dec);
21     /*****
22     Error:
23         void g(auto int a);
24         void f1(void,void);
25         void f2(const auto void);
26     *****/

```

```

27 ty = DeriveType(paramDecl->dec->tyDrvList,
28                               ty, &paramDecl->coord);
29 if (ty != NULL) {
30     ty = AdjustParameter(ty);
31 }
32 if (ty == NULL) {
33     Error(&paramDecl->coord, "Illegal parameter type");
34     return;
35 }
36 if(funcDec->partOfDef &&
37     IsIncompleteType(ty, IGNORE_ZERO_SIZE_ARRAY)) {
38     Error(&paramDecl->coord,
39           "parameter has incomplete type");
40 }
41 id = paramDecl->dec->id;
42 // f(int,int ,int ){}      is illegal.
43 if (id == NULL && funcDec->partOfDef) {
44     Error(&paramDecl->coord, "Expect parameter name");
45     return;
46 }
47 AddParameter(funcDec->sig->params, id, ty,
48               paramDecl->specs->sclass == TK_REGISTER,
49               &paramDecl->coord);
50 }
51 Type AdjustParameter(Type ty) {
52     ty = Unqual(ty);
53     // int f(int arr[3]) ----> int f(int *);
54     if (ty->categ == ARRAY)
55         return PointerTo(ty->bty);
56     if (ty->categ == FUNCTION)
57         return PointerTo(ty);
58     return ty;
59 }

```

图 4.48 CheckParameterTypeList()

对于形参中的数组和函数类型，按 C 标准的规定，我们需要将其调整为指针类型，第 30 行调用的函数 AdjustParameter() 完成了这个工作，与此相关的代码如图 4.48 第 54 至 57 行所示。在函数声明的形参列表中，如果出现不完整的类型（IncompleteType），C 编译器会给出一个警告，例如以下函数声明 f 中的 struct Data。而在函数定义的形参列表中出现的不完整类型，则被视为错误，例如以下函数定义 g 中的 struct Data。其原因是，在不知道 struct Data 的类型信息时，我们无法为形参 b 分配栈空间，而对于函数声明 f 中的形参 a，我们并不需要为之分配栈空间。在 C 语言中，函数声明中的形参名可以省略，而在函数定义中出现的形参名则不能省略。图 4.48 第 36 至 46 行对这些情况进行检查，第 47 行则把形参的类型信息加入到向量 funcDec->sig->params 中。

```

void f(struct Data a);
void g(struct Data b){ //此时 struct Data 还未知，视为 IncompleteType
}
struct Data{
    int a;
};

```

在图 4.48 第 37 行，我们调用 IsIncompleteType() 函数用来判断某个类型是否完整，与其相关的代码在 type.c 中，如图 4.49 所示。图 4.49 第 1 至 5 行的代码用于判断数组类型是否

完整，第 6 至 10 行用于判断枚举类型是否完整，第 11 至 15 行用于检查结构体和联合体类型是否完整。在还未见到结构体、枚举、联合体定义时，我们视这些类型为不完整的，第 9 行和第 14 行的标志位 complete 用于做此标记；而数组大小为 0 时，我们就将其当作不完整类型的。第 16 行的 IsIncompleteType() 函数实际上是分别调用第 1 至 15 行的这 3 个函数来判断。

```

1 int IsZeroSizeArray(Type ty) {
2     ty = Unqual(ty);
3     return (ty->categ == ARRAY &&
4             (((ArrayType)ty)->len == 0) && ty->size == 0);
5 }
6 int IsIncompleteEnum(Type ty) {
7     ty = Unqual(ty);
8     return ( ty->categ == ENUM &&
9             !((EnumType)ty)->complete );
10 }
11 int IsIncompleteRecord(Type ty) {
12     ty = Unqual(ty);
13     return ( IsRecordType(ty) &&
14             !((RecordType) ty)->complete);
15 }
16 int IsIncompleteType(Type ty, int ignoreZeroArray) {
17     ty = Unqual(ty);
18     switch(ty->categ) {
19     case ENUM:
20         return IsIncompleteEnum(ty);
21     case STRUCT:
22     case UNION:
23         return IsIncompleteRecord(ty);
24     case ARRAY:
25         if(ignoreZeroArray) {
26             return IsIncompleteType(ty->bty,
27                                     IGNORE_ZERO_SIZE_ARRAY);
28         } else{
29             if(IsZeroSizeArray(ty)) {
30                 return 1;
31             } else{
32                 return IsIncompleteType(ty->bty,
33                                         !IGNORE_ZERO_SIZE_ARRAY);
34             }
35         }
36     default:
37         return 0;
38     }
39 }
```

图 4.49 IsIncompleteType()

至此，我们讨论了 CheckDeclarationSpecifiers() 和 CheckDeclarator() 这两个函数，为了对经两个函数检查后的结果有个感性认识，我们举以下例子说明。对于 arr1 和 arr2 而言，其声明说明符都为 int，经 CheckDeclarationSpecifiers() 处理后，我们会得到类型信息 T(INT)。

```

int * arr1[5];
int (*arr2)[5];
```

而对于 \*arr1[5] 和 (\*arr2)[5] 来说，图 4.50 给出了语法分析后及经过 CheckDeclarator() 语

义检查后的类型信息链表。结合从声明说明符中得到的类型信息 T(INT)，遍历图 4.50 语义检查后的类型信息链表，依次把类型构造符 POINTER\_TO 或 ARRAY\_OF 作用在已有类型上，我们可得到 arr1 是“array[5] of pointer to int”，即 arr1 是一个长度为 5 的数组，其数组元素的类型为 int \*。而 arr2 是“pointer to array[5] of int”，即 arr2 是一个指针，指向 int[5] 类型的数组。

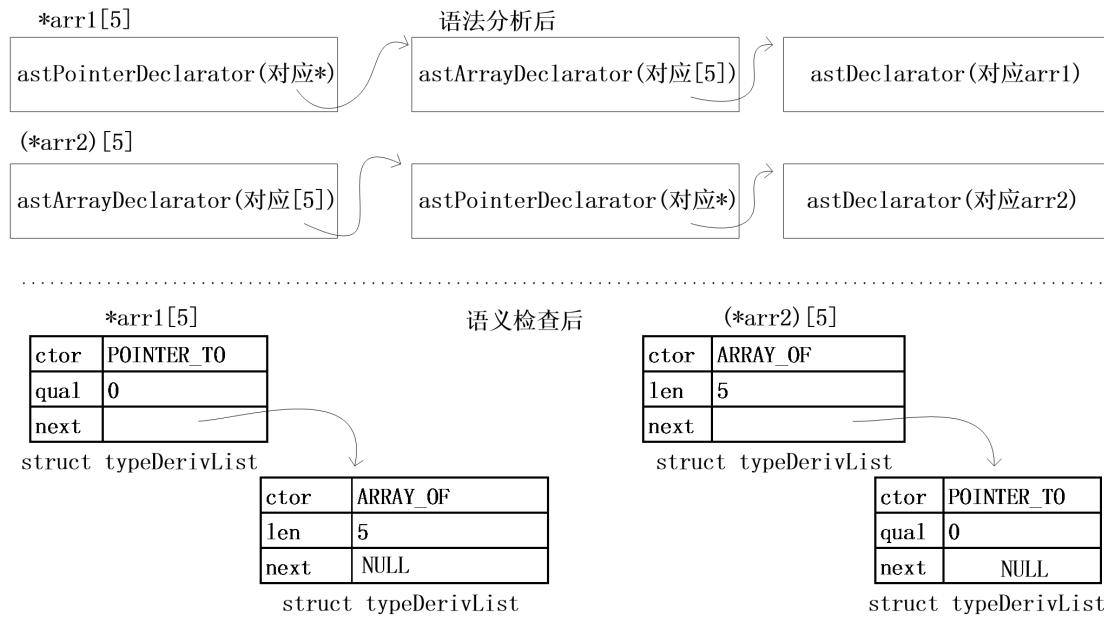


图 4.50 类型信息链表

结合 CheckDeclarationSpecifiers() 函数所得到的类型信息，遍历如图 4.50 中所示的由若干个 struct TypeDerivList 对象构成的链表，我们就可构建起第 2.4 节中介绍的类型结构，与此相关的代码如图 4.51 所示。图 4.51 第 3 至 4 行用于处理指针类型；C 语言中，不允许出现函数的数组，也不允许长度为负数的数组，第 5 至 17 行对这些情况进行检查，第 18 行调用 ArrayOf() 来构建数组的类型结构。而函数的返回值不可以是数组，也不可以是函数类型，第 20 至 27 行对此进行检查，第 28 行调用 FunctionReturn() 来创建函数的类型结构。

```

1 static Type DeriveType(TypeDerivList tyDrvList, Type ty, Coord coord) {
2     while (tyDrvList != NULL) {
3         if (tyDrvList->ctor == POINTER_TO) {
4             ty = Qualify(tyDrvList->qual, PointerTo(ty));
5         } else if (tyDrvList->ctor == ARRAY_OF) {
6             if (ty->categ == FUNCTION) {
7                 Error(coord, "array of function");
8                 ty = PointerTo(ty);
9             }
10            if (IsIncompleteType(ty, !IGNORE_ZERO_SIZE_ARRAY)) {
11                Error(coord, "array has incomplete element type");
12                ((ArrayType) ty)->len = 1;
13            }
14            if (tyDrvList->len < 0) {
15                Error(coord, "size of array is negative");
16                ((ArrayType) ty)->len = 1;
17            }
18            ty = ArrayOf(tyDrvList->len, ty);
19        } else{
20            if (ty->categ == ARRAY) {
21                Error(coord, "function cannot return array type");
22            }
23        }
24    }
25 }

```

```

22             ty = PointerTo(ty);
23         }
24         if( ty->categ == FUNCTION){
25             Error(coord,"function cannot return function type");
26             ty = PointerTo(ty);
27         }
28         ty = FunctionReturn(ty, tyDrvList->sig);
29     }
30     tyDrvList = tyDrvList->next;
31 }
32 return ty;
33 }

```

图 4.51 DeriveType()

经过 DeriveType() 函数的类型推导，我们会为 `int * arr1[5]` 和 `int (*arr2)[5]` 这两个声明分别构建如图 4.52 所示的类型结构。在这小节中，我们重点介绍了 CheckDeclarationSpecifiers()、CheckDeclarator() 和 DeriveType() 这 3 个函数，UCC 编译器用这 3 个函数构建起了类型系统。为了形成整体的概念，在讨论 CheckDeclarationSpecifiers() 函数时，我们有意忽略了为结构体构建类型信息的函数 CheckStructOrUnionSpecifier()。在下一小节中，我们会对此进行分析。

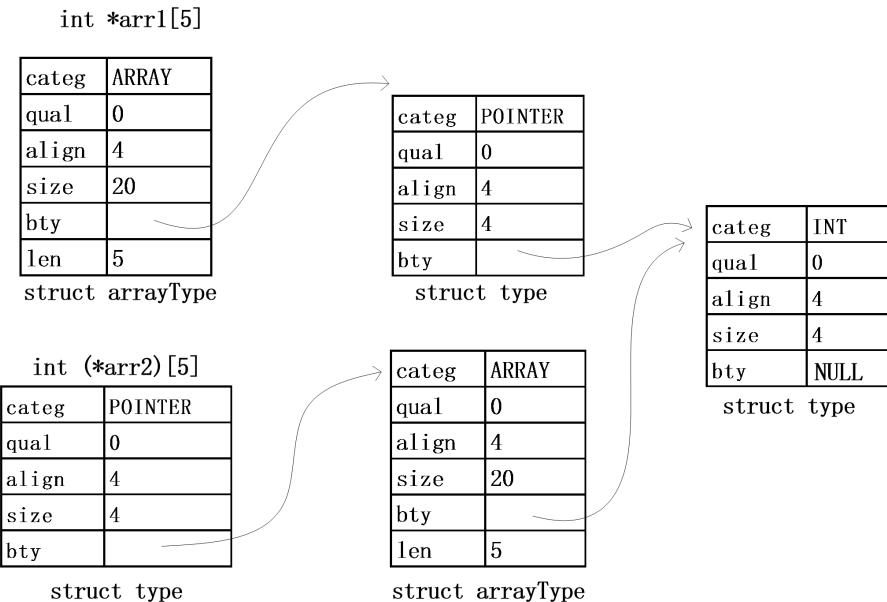


图 4.52 int \* arr1[5] 和 int (\*arr2)[5] 的类型结构

#### 4.4.2 结构体的类型结构

在这一小节中，我们对形如图 4.53 的语法树进行语义检查，从而构建结构体 struct Data 的类型结构。图 4.53 上侧所示的语法树是我们在本小节的起点，而图 4.53 下侧所示的类型结构是我们的目的地。

按照 C 的标准文法，根据“结构体名”和“大括号”的有无情况，“结构或联合说明符”可分为以下 4 种情形，其中第 4 种情况已在语法分析时报错。

- (1) `struct Data1` // 有结构体名但无“大括号”
- (2) `struct {int a; int b;}` // 无结构体名但有“大括号”
- (3) `struct Data2{int a; int b;}` // 有结构体名也有“大括号”
- (4) `struct` // 无结构体名也无“大括号”，语法分析时已报错

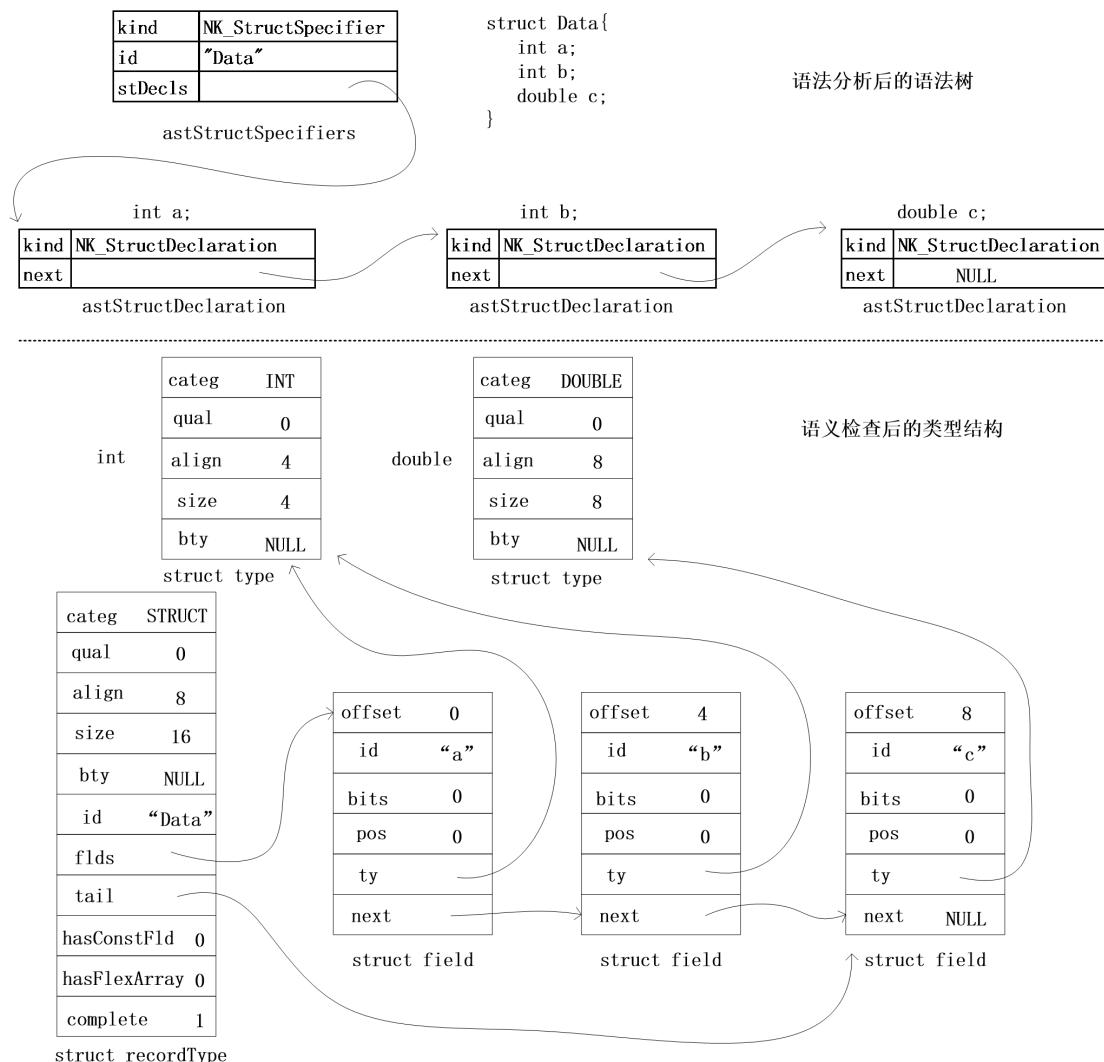


图 4.53 为结构体创建类型结构

如图 4.54 所示的函数 `CheckStructOrUnionSpecifier()`主要是用于处理以上 4 种情况，第 9 至 19 行用于处理形如 `struct Data1` 的“有名无大括号”的情况，第 9 行的 `LookupTag()`用来查询符号表，如果找不到则在第 12 行调用 `StartRecord()` 创建一个如图 4.53 所示的 `struct recordType` 对象，第 13 行的 `AddTag()` 函数用于在符号表中添加一项关于 `struct Data1` 的符号信息，此时结构体 `struct Data1` 的类型还不是完整的，我们暂时只有一个 `struct recordType` 对象，还没有形成如图 4.53 所示的完整类型结构。

```

1 static Type CheckStructOrUnionSpecifier(
2                     AstStructSpecifier stSpec)
3 {
4     int categ =
5         (stSpec->kind == NK_StructSpecifier) ? STRUCT : UNION;
6     Symbol tag;
7     Type ty;
8     AstStructDeclaration stDecl;
9     if (stSpec->id != NULL && !stSpec->hasLbrace) {
10         tag = LookupTag(stSpec->id);
11         if (tag == NULL) {
12             ty = StartRecord(stSpec->id, categ);
13             tag = AddTag(stSpec->id, ty, &stSpec->coord);
14         } else if (tag->ty->categ != categ) {

```

```

15         Error(&stSpec->coord,
16                 "Inconsistent tag declaration.");
17     }
18     return tag->ty;
19 }
20 else if (stSpec->id == NULL && stSpec->hasLbrace) {
21     ty = StartRecord(NULL, categ);
22     ((RecordType)ty)->complete = 1;
23     goto chk_decls;
24 }else if (stSpec->id != NULL && stSpec->hasLbrace) {
25     tag = LookupTag(stSpec->id);
26     if (tag == NULL || tag->level < Level){
27         ty = StartRecord(stSpec->id, categ);
28         ((RecordType)ty)->complete = 1;
29         AddTag(stSpec->id, ty,&stSpec->coord);
30     }else if (tag->ty->categ == categ
31                 && IsIncompleteRecord(tag->ty)){
32         ty = tag->ty;
33         ((RecordType)ty)->complete = 1;
34     }else{
35         if(tag->ty->categ != categ){
36             Error(&stSpec->coord,
37                     "\'%s\'defined as wrong kind of tag.",
38                     stSpec->id);
39         }else{
40             Error(&stSpec->coord,
41                     "redefinition of \'%s %s\'.",
42                     GetCategName(categ),stSpec->id);
43         }
44     return tag->ty;
45 }
46     goto chk_decls;
47 } else{
48     ty = StartRecord(NULL, categ);
49     EndRecord(ty,&stSpec->coord);
50     return ty;
51 }
52 chk_decls:
53 stDecl = (AstStructDeclaration)stSpec->stDecls;
54 while (stDecl){
55     CheckStructDeclaration(stDecl, ty);
56     stDecl = (AstStructDeclaration)stDecl->next;
57 }
58 EndRecord(ty,&stSpec->coord);
59 return ty;
60 }

```

图 4.54 CheckStructOrUnionSpecifier()

对于“无名但有大括号”的情况，第 20 至 23 行会进行预处理，第 22 行置 complete 为 1，这意味着当 CheckStructOrUnionSpecifier() 函数返回时，我们就能得到一个完整的结构体类型结构。而真正构成如图 4.53 类型结构的工作，则在图 4.54 第 52 至 59 行来完成。结构体中的成员域在 C 标准文法中被称为 StructDeclaration，第 55 行调用 CheckStructDeclaration() 来对成员域进行检查，由此获得的成员域类型信息会存放在如图 4.53 所示的 struct filed 对象

中。由图 4.54 第 54 行的 while 循环得到各成员域的类型信息后，我们还需要进行综合，从而计算出各成员域在整个结构体中的偏移信息 offset，这些信息最终决定了结构体对象在内存中的布局，第 58 行的 EndRecord() 函数完成了这个工作。第 24 至 46 行的代码用于对“有名且有大括号”的情况进行预处理。

简而言之，构建如图 4.53 类型结构的主要步骤有：

- (1) 调用 StartRecord() 函数构建一个 struct recordType 对象；
- (2) 为各个成员域调用 CheckStructDeclaration() 函数来创建相应的 struct filed 对象；
- (3) 调用 EndRecord() 函数来计算结构体的内存布局，主要是各成员域的偏移位置。

下面，我们就依次来对相关函数进行讨论，图 4.55 给出了 StartRecord() 函数的代码，第 3 行在堆空间中创建了一个 struct recordType 对象，第 5 至 8 行对其进行初始化，第 7 行的 rty->flds 指向若干个 struct filed 对象构成的链表，第 8 行置 complete 标志位为 0，表示这还是一个不完整的结构体类型。通过 CheckStructDeclaration() 函数获得的成员域类型信息会存放在一个 struct filed 对象中，可调用第 11 行的 AddField() 函数，把 struct filed 对象添加到 struct recordType 对象中。

```

1 Type StartRecord(char *id, int categ) {
2     RecordType rty;
3     ALLOC(rty);
4     memset(rty, 0, sizeof(*rty));
5     rty->categ = categ;
6     rty->id = id;
7     rty->tail = &rty->flds;
8     rty->complete = 0;
9     return (Type)rty;
10 }
11 Field AddField(Type ty,
12                  char *id, Type fty, int bits) {
13     RecordType rty = (RecordType)ty;
14     Field fld;
15     if (fty->size == 0) {
16         if (fty->categ == ARRAY) {
17             rty->hasFlexArray = 1;
18         }
19     }
20     if (fty->qual & CONST) {
21         rty->hasConstFld = 1;
22     }
23     ALLOC(fld);
24     fld->id = id;
25     fld->ty = fty;
26     fld->bits = bits;
27     fld->pos = fld->offset = 0;
28     fld->next = NULL;
29     *rty->tail = fld;
30     rty->tail = &(fld->next);
31     return fld;
32 }
33 Field LookupField(Type ty, char *id) {
34     RecordType rty = (RecordType)ty;
35     Field fld = rty->flds;
36     while (fld != NULL) {

```

```

37     if (fld->id == NULL &&
38             IsRecordType(fld->ty)) {
39         Field p;
40         p = LookupField(fld->ty, id);
41         if (p) {
42             return p;
43         }
44     }
45     else if (fld->id == id) {
46         return fld;
47     }
48     fld = fld->next;
49 }
50 return NULL;
51 }

```

图 4.55 StartRecord()

C 语言中，允许在结构体的末尾处定义变长的数组，如以下 struct Packet 所示，图 4.55 第 17 行的 hasFlexArray 用于记录结构体中是否有变长数组。当结构体中有一个成员域含有 const 限定符时，则整个结构体对象要被视为 const，第 21 行置标志位 hasConstFld 为 1，第 23 至 28 行创建了一个 struct filed 对象并进行相应的初始化，第 29 和 30 行把这个对象加入 rty->flds 所指向的链表中。

```

struct Packet{
    int len;
    char data[];
};

```

图 4.55 第 33 行的 LookupField() 函数用来检索在 struct recordType 对象中是否存在名为 id 的成员域，当成员域没有名字而且其类型也是无名结构体时，此时我们需要在第 40 行递归调用 LookupField() 函数，在这个“无名结构体”中进行查找，这用于处理如下所示的 dt.b。

```

struct Data{
    struct { // 无名结构体，若改为 struct ABC，则导致 dt.b 为语法错误
        int a;
        int b;
    };      // 无名域成员
    int c;
}
struct Data dt;
dt.b;

```

需要注意的是，如果把上述“无名结构体”改成有名的，例如改为 struct ABC，则由于在 C 语言中，struct ABC 并不被当成 struct Data 的“内部类”，两者所处作用域是相同的，这会导致 dt.b 被视为语法错误，此时，C 编译器认为在 struct Data 中并不存在名为 b 的成员域。

接下来，我们来看一下对结构体成员域进行检查的函数 CheckStructDeclaration()，如图 4.56 所示。对于结构体成员域中的“声明说明符”，我们在第 28 行调用函数 CheckDeclarationSpecifiers() 来获取其中的类型信息，对于形如第 33 行的多个声明符，我们在第 36 至 39 行的 while 循环中调用函数 CheckStructDeclarator() 来依次处理。

```

1 static void CheckStructDeclarator(
2     Type rty, AstStructDeclarator stDec, Type fty) {
3     char *id = NULL;
4     int bits = 0;
5     if (stDec->dec != NULL) {

```

```

6     CheckDeclarator(stDec->dec);
7     id = stDec->dec->id;
8     fty = DeriveType(  stDec->dec->tyDrvList,
9                         fty, &stDec->coord);
10    }
11    .....
12 /*****
13 Error examples:
14     struct Data{
15         int arr[];
16         double len;
17         int len;
18         int low:"123";
19         double high:100;
20         void f(void);
21     };
22 ****/
23 AddField(rty, id, fty, bits);
24 }
25 static void CheckStructDeclaration(
26 AstStructDeclaration stDecl, Type rty){
27 AstStructDeclarator stDec;
28 CheckDeclarationSpecifiers(stDecl->specs);
29 stDec = (AstStructDeclarator)stDecl->stDecls;
30 .....
31 ****/
32     struct Data{
33         int c, d;
34     };
35 ****/
36 while (stDec){
37     CheckStructDeclarator(rty, stDec, stDecl->specs->ty);
38     stDec = (AstStructDeclarator)stDec->next;
39 }
40 }

```

图 4.56 CheckStructDeclaration()

函数 CheckStructDeclarator()的代码如图 4.56 第 1 至 24 行所示，在第 6 行我们调用 CheckDeclarator()来获取声明符中的类型信息，第 8 行通过调用 DeriveType()来对“声明说明符”和“声明符”这两部分的类型信息进行综合，最后通过第 23 行的 AddField()函数把成员域的类型信息添加到结构体的类型结构中，从而构成图 4.53 所示的类型结构。在第 12 至 22 行的注释中，我们给出了结构体定义的一些错误例子，在函数 CheckStructDeclarator()中，我们需要检测这些错误，相关代码并不太复杂，这里从略。由此可以发现，我们最终仍然是按照 CheckDeclarationSpecifiers()、CheckDeclarator()和 DeriveType()这三部曲，来为结构体成员域构建类型结构。

由于已经知道了结构体中各成员的类型信息，我们就能知道各成员域所要占的内存大小，而且成员域是依次进行定义的，由此可计算其在结构体对象中的偏移位置，这个工作由 EndRecord()函数来完成。只是碰到“无名结构体”时，会相对麻烦一些，如图 4.57 第 3 至第 7 行代码所示。

```

1 struct Data{
2     int a;      // offset is 0
3     struct{

```

```

4         int b1; //0
5         int b2; //4
6         int b3; //8
7     };           // offset is 4
8     double c;   // offset is 16
9     struct{
10         int d1; // 0
11         int d2; // 4
12         int d3; // 8
13     }d;           //
14 };
15
16 void AddOffset(RecordType rty, int offset){
17     Field fld = rty->flds;
18     while (fld) {
19         fld->offset += offset;
20         if (fld->id == NULL && IsRecordType(fld->ty)){
21             AddOffset((RecordType)fld->ty, fld->offset);
22         }
23         fld = fld->next;
24     }
25
26 }

```

图 4.57 AddOffset()

在 C 语言中，对于以下代码而言，当我们使用 `dt.b3` 时，我们需要知道 `b3` 在整个 `struct Data` 中的偏移，由图 4.57 第 2 行，我们可以发现成员域 `a` 的偏移为 0，其大小为 4，之后的偏移为 4，而第 6 行的 `b3` 在无名结构体中的偏移为 8，两者相加，则可以得到 `b3` 在整个结构体 `struct Data` 中的偏移为 12。但是对于 `dt.d.d3` 来说，我们只要计算出第 13 行的 `d` 在 `struct Data` 的偏移，再计算出 `d3` 在第 9 行所对应的无名结构体的偏移即可，在中间代码生成时，我们先处理 `dt.d`，之后再处理(`dt.d`).`d3`。

```

struct Data dt;
dt.b3;
dt.d.d3;

```

图 4.57 第 16 至 26 行的 `AddOffset()` 函数用于为形如第 3 至 7 行的“无名结构体”计算各成员域的偏移，考虑到“无名结构体”内可能还有嵌套的“无名结构体”，我们需要在第 21 行递归地调用 `AddOffset()` 来计算相应的偏移。对于图 4.57 第 9 至 13 行的“无名结构体”，由于成员域 `d` 是有名的，所以我们不需要调用 `AddOffset()` 来计算 `d1`、`d2` 和 `d3` 在整个 `struct Data` 中的偏移。

在此基础上，我们来分析一下 `EndRecord()` 函数，为了能更清楚地看到整个代码的总体流程，我们在图 4.58 第 13 至 21 行忽略了对于结构体位域成员的处理，第 6 至 31 行用于处理结构体类型，而第 32 至 46 行则用于处理联合体类型。对于联合体而言，我们要遍历形如图 4.53 中的 `struct field` 对象链表，找出其中占最大内存空间的成员，此成员的大小也就是整个联合体对象要占内存的大小。联合体中不可以有变长数据，第 42 至 46 行对此进行检查。

```

1 void EndRecord(Type ty, Coord coord){
2     RecordType rty = (RecordType)ty;
3     Field fld = rty->flds;
4     int bits = 0;
5     int intBits = T(INT)->size * 8;
6     if (rty->categ == STRUCT){
7         while (fld) {

```

```

8         fld->offset = rty->size =
9             ALIGN(rty->size, fld->ty->align);
10        if (fld->id == NULL && IsRecordType(fld->ty)){
11            AddOffset((RecordType)fld->ty, fld->offset);
12        }
13        /*****
14         struct B{
15             int a1:12;
16             int b1;      // (1)
17             int a2:12;
18             int b2:12;  // (2)
19             int b3:20;  // (3)
20         };
21         *****/
22        if (fld->ty->align > rty->align){
23            rty->align = fld->ty->align;
24        }
25        fld = fld->next;
26    }
27    if (bits != 0){
28        rty->size += T(INT)->size;
29    }
30    rty->size = ALIGN(rty->size, rty->align);
31 }
32 else{      //UNION
33     while (fld){
34         if (fld->ty->align > rty->align){
35             rty->align = fld->ty->align;
36         }
37         if (fld->ty->size > rty->size){
38             rty->size = fld->ty->size;
39         }
40         fld = fld->next;
41     }
42     if(rty->hasFlexArray){
43         Error(coord,
44                 "flexible array member in union");
45     }
46 }
47 if(rty->categ == STRUCT &&
48     rty->size == 0 && rty->hasFlexArray){
49     Error(coord,
50             "flexible array member in empty struct");
51 }
52 }

```

图 4.58 EndRecord()

而对于结构体而言，整个结构体对象所占内存空间的大小，至少是各成员域所占内存大小的总和，如果考虑到成员域的对齐，则有时还要偏大一点。例如在以下结构体 struct Data 中，各成员域所占内存大小的总和为 5 字节，但考虑到 int 型一般是按 4 字节来对齐的，所以在成员 ch 和 num 之间会有 3 字节的空白，对齐后的整个 struct Data 所占内存大小为 8 字节。图 4.58 第 7 至 26 行的 while 循环，会对各成员域所占内存大小进行累加，并在第 8 行得到当前成员域的偏移，在第 9 行用 ALIGN 宏来进行对齐。对于结构体内的“无名结构体”，

我们在第 11 行调用前文讨论过的 AddOffset() 函数，从而计算“无名结构体”内的成员域在整个结构体中的偏移。

```
struct Data{
    char ch;      // 占 1 字节
    int num;      // 占 4 字节
};
```

而对于如下结构体 struct Buffer 而言，由于整个结构体 struct Buffer 中只有一个变长数组，这被视为非法的，我们会在第 47 至 51 行对此进行检查。

```
struct Buffer{
    char buf[];
};
```

与图 4.58 第 13 至 21 行注释所对应的代码，主要用于处理以下情况：

(1) 如图 4.58 第 16 行所示，当前成员 b1 不是位域，此时，前一个位域 a1 虽然只有 12 位，但我们让其独占 4 个字节，即一个 int 所占的内存大小；

(2) 如图 4.58 第 18 行所示，当前成员 b2 是位域，且能够放置在一个 int 剩余的位空间中，例如此处 a2 占去了 12 位，还剩 20 位，足够 b2 存放；

(3) 如图 4.58 第 19 行所示，当前成员 b3 是位域，但一个 int 剩余的位空间已经不够存放，a2 和 b2 共占去了 24 位，剩下的 8 位不够存放 b3。此时我们可以新开辟一个 int 来存放 b3 的 20 位数据。

总之，构建类型结构的三部曲 CheckDeclarationSpecifiers()、CheckDeclarator() 和 DeriveType() 是最关键的地方。在此基础上去理解对枚举进行检查的 CheckEnumSpecifier() 函数，及用于检查 typedef 自定义类型名的函数 CheckTypedef()，则相对容易，我们就不再啰嗦。

#### 4.4.3 结构体和数组的初始化

在这一小节中，我们来讨论一下结构体和数组的初始化。在不少程序设计语言的教材中，对变量的声明和定义是有严格区分的。但从 C 标准文法的角度来看，它们都是由非终结符 declaration 产生的，因此对变量初始化的语义检查也在 declchk.c 中进行。我们先举一个例子来说明“我们在这一小节所处的起点和终点”，如图 4.59 所示。在图的左侧，我们给出初值 {{20},30} 经语法分析后形成的语法树，这是我们本小节的起点。在图的右侧，由 struct initData 对象构成的链表则是我们在本小节的终点。在对左侧语法树进行语义检查后，dt.a 在结构体对象 dt 中的偏移为 0，而 dt.c 的偏移为 8，因此图中 offset 为 0 的结点代表“我们需要把 dt.a 初始化为 20”，而 offset 为 8 的结点代表“我们需要把 dt.c 初始化为 30”。

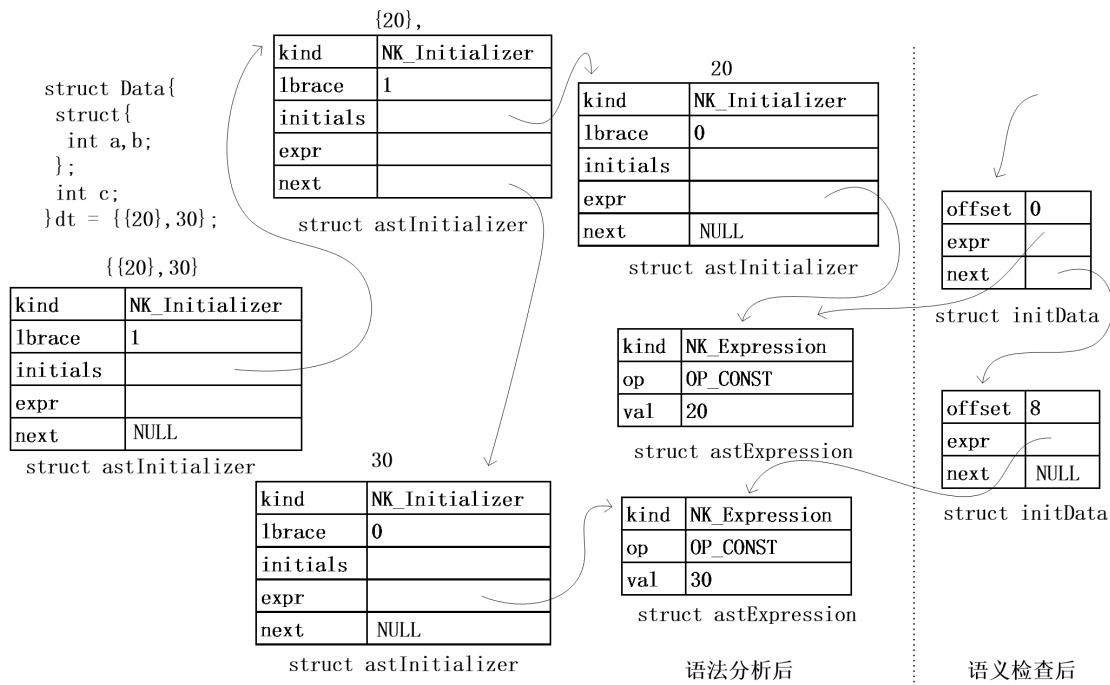


图 4.59 struct initData

结构体中允许存在位域成员，其初始化需要经过一些位运算，如图 4.60 第 1 至 8 行注释中的代码所示，对于第 7 行的 pte 而言，其实际所占的内存为 4 字节，我们需要先把初值 3 左移 12 位后，再跟初值 64 进行或运算，然后用所得结果( $(3 \ll 12) | 64$ )来初始化 pte 对象。图 4.60 第 27 至 45 行的 PlaceBitField() 函数实现了左移运算，对于常量，在编译时我们就可以在第 31 至 34 行进行常量折叠，而对于不是常量的操作数，才需要在第 35 至 43 行生成一个进行左移运算的语法树结点。而图 4.60 第 9 至 26 行的 BORBitField() 函数实现了按位或运算。

```

1  ****
2  struct PTE{
3      int offset:12;
4      int pgNo:20;
5  };
6  // (3 << 12) | 64
7  struct PTE pte = {64,3};
8  ****
9  static AstExpression BORBitField(
10      AstExpression expr1, AstExpression expr2){
11      AstExpression bor;
12
13      if (expr1->op == OP_CONST
14          && expr2->op == OP_CONST){
15          expr1->val.i[0] |= expr2->val.i[0];
16          return expr1;
17      }
18      CREATE_AST_NODE(bor, Expression);
19      bor->coord = expr1->coord;
20      bor->ty = expr1->ty;
21      bor->op = OP_BITOR;
22      bor->kids[0] = expr1;
23      bor->kids[1] = expr2;
24

```

```

25   return bor;
26 }
27 static AstExpression PlaceBitField(
28     Field fld, AstExpression expr) {
29   AstExpression lsh;
30   union value val;
31   if (expr->op == OP_CONST) {
32     expr->val.i[0] <= fld->pos;
33     return expr;
34   }
35   CREATE_AST_NODE(lsh, Expression);
36   lsh->coord = expr->coord;
37   lsh->ty = expr->ty;
38   lsh->op = OP_LSHIFT;
39   lsh->kids[0] = expr;
40   val.i[1] = 0;
41   val.i[0] = fld->pos;
42   lsh->kids[1] = Constant(expr->coord, T(INT), val);
43
44   return lsh;
45 }

```

图 4.60 BORBitField()

接下来，我们来看一下对初值进行语义检查的函数 CheckInitializer()，如图 4.61 所示。图 4.61 第 8 至第 18 行的代码，用于处理形如第 9 行注释所示的数组初始化，“用字符串来初始化一维数组”是数组初始化的一个特例。图 4.61 第 10 行调用 CheckCharArrayInit() 函数来处理一些非法的情况。函数 CheckCharArrayInit() 的代码相对简单，我们就不再展开，只在图 4.61 第 12 至 14 行注释中给出一些错误示例。而对于结构体或联合体对象而言，其初值也可以是同类型的其他对象，如图第 22 至 23 行所示，第 26 行的 CanAssign() 函数可用于判断第 23 行 dt2 和 dt 的类型是否匹配，我们已在前面的章节中分析过这个函数，第 29 至 32 行则生成一个 struct initData 对象来记录这个初值。

```

1 static void CheckInitializer(AstInitializer init, Type ty)
2 {
3   int offset = 0, error = 0;
4   struct initData header;
5   initData tail = &header;
6   initData prev, curr;
7   header.next = NULL;
8   if(ty->categ == ARRAY && !init->lbrace) {
9     // char arr[] = "abcdef";
10    if(!CheckCharArrayInit(init->expr, ty->bty)) {
11      ****
12      int arr1[] = "abc";           // Linux
13      unsigned short arr2[] = "abc"; // windows
14      char arr3[] = L"abc";
15      ****
16      return ;
17    }
18  }
19  else if ((ty->categ == STRUCT || ty->categ == UNION)
20    && ! init->lbrace) {
21  ****

```

```

22         struct Data dt;
23         struct Data dt2 = dt;
24         ****
25         init->expr = Adjust(CheckExpression(init->expr), 1);
26         if (!CanAssign(ty, init->expr)){
27             Error(&init->coord, "Wrong initializer");
28         }else{
29             ALLOC(init->iidata);
30             init->iidata->expr = init->expr;
31             init->iidata->offset = 0;
32             init->iidata->next = NULL;
33         }
34     return;
35 }
36
37 CheckInitializerInternal(
38     &tail, init, ty, &offset, &error);
39 if (error)
40     return;
41 init->iidata = header.next;
42 prev = NULL;
43 curr = init->iidata;
44 while (curr){
45     if (prev != NULL && prev->offset == curr->offset){
46         prev->expr = BORBitField(prev->expr, curr->expr);
47         prev->next = curr->next;
48         curr = curr->next;
49     } else{
50         prev = curr;
51         curr = curr->next;
52     }
53 }
54 }
```

图 4.61 CheckInitializer()

图 4.61 第 37 行调用的 CheckInitializerInternal() 函数才是真正对初值进行语义检查的核心函数，此函数返回后，我们会得到如图 4.59 所示的由若干个 struct initData 对象构成的链表。对于如图 4.60 第 7 行所示的初值，我们需要进行按位或运算，从而得到初值( $(3<<12)|64$ )，图 4.61 第 41 至 53 行的代码用于对此进行处理，我们在第 46 行调用函数 BORBitFiled() 进行按位或运算，之后在第 47 行删除了链表中多余的 struct initData 对象。

需要结合图 4.59 所示的语法树来分析函数 CheckInitializerInternal()，其代码如图 4.62 所示。图 4.62 第 7 至 38 用于对整型、浮点型和指针类型等标量类型的初始化进行检查，第 39 至 55 行则用于对联合体对象的初始化进行语义检查；而在第 56 至 58 行，限于篇幅，我们暂时省略了用于处理数组和结构体初始化的代码。图 4.62 第 7 行判断被初始化对象的类型是否为标量类型，第 9 至 13 行的 while 循环用于处理形如 $\{\{3\}\}$ 的标量初始化值，此时我们给出一个“warning:braces around scalar initializer”警告，表示此处大括号是多余的。第 15 行调用 CanAssign() 来判断一下能否把初值赋值给被初始化对象，第 28 行调用 Cast() 进行必要的类型转换。但对于第 20 至 22 行注释中所示的整型与指针类型之间的初始化，由于在 32 位机器中，指针类型和 int 型占用相同大小的存储空间，此时我们就不必构造进行类型转换的语法树结点，第 25 行的 if 语句会对此进行判断。第 31 至 36 行则创建一个 struct initData 对象，并将其加入由 struct initData 对象构成的链表的末尾处。

```

1 static AstInitializer CheckInitializerInternal(
2     initData *tail, AstInitializer init, Type ty,
3     int *offset, int *error){
4     AstInitializer p;
5     int size = 0;
6     initData initd;
7     if (IsScalarType(ty)){
8         p = init;
9         while(p->lbrace){ // int a = {{3}};
10            Warning(&p->coord,
11                  "braces around scalar initializer");
12            p = (AstInitializer) p->initials;
13        }
14        p->expr = Adjust(CheckExpression(p->expr), 1);
15        if (! CanAssign(ty, p->expr)){
16            Error(&init->coord, "Wrong initializer");
17            *error = 1;
18        }else{
19            ****
20            int num;
21            int addr = &num;
22            int *ptr = 0x8000;
23            ****
24            Type lty = ty, rty = p->expr->ty;
25            if( !(IsPtrType(lty) && IsIntegType(rty)
26                || IsPtrType(rty) && IsIntegType(lty)) &&
27                (lty->size == rty->size))){
28                p->expr = Cast(ty, p->expr);
29            }
30        }
31        ALLOC(initd);
32        initd->offset = *offset;
33        initd->expr = p->expr;
34        initd->next = NULL;
35        (*tail)->next = initd;
36        *tail = initd;
37        return (AstInitializer) init->next;
38    }
39    else if (ty->categ == UNION){
40        p = init->lbrace ? (AstInitializer) init->initials : init;
41        ty = ((RecordType)ty)->flds->ty;
42        p = CheckInitializerInternal(tail, p, ty, offset, error);
43        if (init->lbrace){
44            if (p != NULL) {
45                ****
46                union {
47                    int a;
48                    double b;
49                } dt = {3,2.0};
50                ****
51                Warning(&init->coord, " excess elements in union initializer");
52            }
53            return (AstInitializer) init->next;

```

```

54         }
55         return p;
56     }else if (ty->categ == ARRAY) { ...
57     }else if (ty->categ == STRUCT) { ...
58     }
59     return init;
60 }

```

图 4.62 CheckInitializerInternal()

而对于联合体对象的初始化，按 C 标准的规定，我们只对联合体中最先出现的成员进行初始化，由第 41 行可获得该成员的类型。第 40 行的作用是“去掉初值中可能存在的大括号”，第 42 行递归地调用 CheckInitializerInternal() 函数来进行语义检查。对于图 4.62 第 46 至 49 行所示的初值来说，其中的 2.0 是多余的，此时我们在第 51 行给出一个警告。对于如下所示的联合体初始化而言，在图 4.62 第 42 行递归调用的 CheckInitializerInternal() 函数中，我们需要对字符数组 char buf[16] 进行初始化。

```

union {
    char buf[16];
    int num;
} dt = {"Hello World.\n"};

```

接下来，我们来看一下对数组初始化进行语义检查的相关代码，如图 4.63 所示。对于形如第 4 行注释所示的“用字符串来初始化一维数组”的情况，我们在第 5 至 33 行进行特殊处理，完成处理后直接从第 33 行返回。对于其他形式的数组初始化，则在第 35 至 42 行的 while 循环中递归地调用函数 CheckInitializerInternal()，对每个数组元素进行初始化。图 4.63 第 43 至 50 行的代码，主要是用于处理形如第 44 行注释所示的未指定大小的数组，在检查完初值 {1,2,3,4,5} 后，可以由初值计算出其数组所占内存的大小。而对于如图第 52 行注释中所示的数组初始化，其中的 4 是多余的，我们会在第 53 行给出一个警告。

```

1  if (ty->categ == ARRAY){//CheckInitializerInternal
2      int start = *offset;
3      p = init->lbrace ? (AstInitializer)init->initials : init;
4      // char buf[] = {"123"}; 或者 char buf[] = "123";
5      if (((init->lbrace && ! p->lbrace && p->next == NULL)
6          || ! init->lbrace)
7          && p->expr->op == OP_STR
8          && ty->bty->categ / 2 == p->expr->ty->bty->categ / 2
9          && ty->bty->categ != ARRAY) {
10         size = p->expr->ty->size;
11         if(ty->size == 0){// char buf[] = "123";
12             ArrayType aty = (ArrayType)ty;
13             aty->size = size;
14             if(aty->bty->size != 0){
15                 aty->len = size / aty->bty->size;
16             }
17         }
18         else if (ty->size == size - p->expr->ty->bty->size) {
19             // char str1[3] = "123";
20             p->expr->ty->size = size - p->expr->ty->bty->size;
21         }
22         else if (ty->size < size){
23             // char str2[3] = "123456";
24             p->expr->ty->size = ty->size;
25             Warning(&init->coord,"..... too long");

```

```

26      }
27      ALLOC(initd);
28      initd->offset = *offset;
29      initd->expr = p->expr;
30      initd->next = NULL;
31      (*tail)->next = initd;
32      *tail = initd;
33      return (AstInitializer) init->next;
34  }
35 while (p != NULL){
36     p = CheckInitializerInternal(
37         tail, p, ty->bty, offset, error);
38     size += ty->bty->size;
39     *offset = start + size;
40     if (ty->size == size)
41         break;
42  }
43 if (ty->size == 0){
44     // int arr[] = {1,2,3,4,5};
45     ArrayType aty = (ArrayType) ty;
46     if(aty->bty->size != 0 && aty->len == 0){
47         aty->len = size/aty->bty->size;
48     }
49     ty->size = size;
50  }
51 if (init->lbrace){
52     if(p){ // int arr2[3] = {1,2,3,4};
53         Warning(&p->coord, "excess elements ...");
54     }
55     return (AstInitializer) init->next;
56  }
57 return p;
58 }

```

图 4.63 CheckInitializerInternal()\_ARRAY

图 4.63 第 5 行的 if 语句主要是用于判断对字符数组的初始化是否合法，C 语言的字符串可以是“abc”，也可以宽字符 L “abc”，我们不可以用“abc”来初始化宽字符数组，也不可以用 L “abc”来初始化 char 字符数组，第 7 至 9 行的条件对此进行限制。而第 5 至 6 行的条件则要求用于初始化数组的字符串应形如“abc”或{“abc”}，但不可以是{“abc”，123}。如果 C 程序员没有指定字符数组的大小，如图 4.63 第 11 行注释所示，则在第 15 行计算出其数组元素的个数。如果 C 程序员指定的数组元素个数正好和字符串长度一样，我们就舍弃字符串末尾的'\0'字符，第 18 至 21 行对此进行处理。对于形如 char str2[3] = “123456”的数组容量不够的情况，我们在第 24 行舍弃多余的字符，同时在第 25 行给出一个警告。第 27 至 32 行则创建一个 struct initData 对象，并插入到链表的末尾。

对结构体进行初始化的代码如图 4.64 所示，对结构体中的各个成员域，我们在第 19 行递归地调用 CheckInitializerInternal() 函数来处理，如果是位域成员，我们需要在第 22 行调用 PlaceBitFiled() 函数进行必要的左移运算，例如图 4.60 第 6 行的(3<<12)。对于形如第 29 行的初始化，由于 4 和 5 是多余的，我们会在第 30 行给出一个警告。

```

1 if(ty->categ == STRUCT){
2     int start = *offset;
3     Field fld = ((RecordType)ty)->flds;

```

```

4     p = init->lbrace ? (AstInitializer)init->initials : init;
5     while (fld && p){
6         *offset = start + fld->offset;
7         if( (IsRecordType(fld->ty) && fld->id == NULL)) {
8             /*****struct *****
9             struct {
10                 int a;      // offset ---- 0
11                 struct{
12                     int b;    // offset ---- 4
13                 };
14                 int c;      // offset ---- 8
15             }dt = {1,2,3};
16             *****/
17             *offset = start;
18         }
19         p = CheckInitializerInternal(
20             tail, p, fld->ty, offset, error);
21         if (fld->bits != 0){
22             (*tail)->expr = PlaceBitField(fld, (*tail)->expr);
23         }
24         fld = fld->next;
25     }
26     *offset = start + ty->size;
27     if (init->lbrace){
28         if (p != NULL) {
29             // struct {int a;int b;int c;} dt = {1,2,3,4,5};
30             Warning(&init->coord,
31                     "excess elements in struct initializer");
32         }
33         return (AstInitializer)init->next;
34     }
35     return (AstInitializer)p;
36 }

```

图 4.64 CheckInitializerInternal()\_STRUCT

按 C 标准的规定，用于初始化全局变量或静态变量的初值应是常量，这样的好处是在 C 程序开始运行时，我们只需要从外存（或者从 ROM）中加载初值映像到内存即可，不需要执行额外的代码来计算初值。C++语言放宽了这个限制，在 C++中以下代码是合法的，即我们可以在运行时调用一个函数 f() 来计算全局变量 number 的初值，这也意味着函数 f() 会比 main() 函数更早执行。但在 C 语言中，以下代码是非法的，其原因就在于 C 标准要求全局变量的初值为常量。

```

int f(void) {
    //...
}

```

```
int number = f();
```

但对于以下函数 g 中的局部变量 local 来说，由于其所处栈空间是在运行时动态分配的，我们可以调用函数 f() 来对 local 进行初始化。这在 C 和 C++ 中都是合法的。

```

void g(void){
    int local = f();
}

```

由于有这样语义上的差别，在调用函数 CheckInitializer() 完成对初值的检查后，对于全

局变量和静态变量，我们还需要再检查一下其初值是否为常量，图 4.65 中的函数 CheckInitConstant() 用于此目的。图 4.65 第 3 行的 while 循环用于遍历由若干个 struct initData 对象构成的链表，第 4 行的 if 条件用于检查初值是否为常量，这包括 op 域为 OP\_CONST 的语法树结点(例如整型或浮点型常数)、字符串常量 OP\_STR 和地址常量。对于地址常量的检查由第 14 行的函数 CheckAddressConstant() 来完成。地址常量的类型应是指针类型，第 18 行对此进行判断。由于 C 语言存在指针运算，对于形如 ptr+n 或 ptr-n 的指针运算，如果 ptr 是地址常量，且 n 为常量，则 ptr+n 和 ptr-n 是地址常量，这个规则可递归地应用到表达式(ptr+n)+k 上，我们最终可得到常量 ptr + (n+k)，第 20 至 28 行的 if 语句用于对此进行处理。

```

1 static void CheckInitConstant(AstInitializer init){
2     initData initd = init->idata;
3     while (initd) {
4         if (! (initd->expr->op == OP_CONST
5                 || initd->expr->op == OP_STR
6                 || (initd->expr =
7                     CheckAddressConstant(initd->expr)))) {
8             Error(&init->coord, "Initializer must be constant");
9         }
10        initd = initd->next;
11    }
12    return;
13 }
14 static AstExpression CheckAddressConstant(AstExpression expr) {
15     AstExpression addr;
16     AstExpression p;
17     int offset = 0;
18     if (! IsPtrType(expr->ty))
19         return NULL;
20     if (expr->op == OP_ADD || expr->op == OP_SUB) {
21         addr = CheckAddressConstant(expr->kids[0]);
22         if (addr == NULL || expr->kids[1]->op != OP_CONST)
23             return NULL;
24         expr->kids[0] = addr->kids[0];
25         expr->kids[1]->val.i[0] +=
26             (expr->op == OP_ADD ? 1 : -1) * addr->kids[1]->val.i[0];
27         return expr;
28     }
29     if (expr->op == OP_ADDRESS) { // &num
30         addr = expr->kids[0];
31     } else{ // arr
32         addr = expr;
33     }
34     while (addr->op == OP_INDEX || addr->op == OP_MEMBER) {
35         if (addr->op == OP_INDEX){ // arr[3][4]
36             if (addr->kids[1]->op != OP_CONST) {
37                 return NULL;
38             }
39             offset += addr->kids[1]->val.i[0];
40         } else{ // a.b.c
41             Field fld = addr->val.p;
42             offset += fld->offset;
43         }
44         addr = addr->kids[0];
}

```

```

45 }
46 if (addr->op != OP_ID || (expr->op != OP_ADDRESS
47             && ! addr->isarray && ! addr->isfunc))
48     return NULL;
49 ((Symbol)addr->val.p)->ref++;
50 CREATE_AST_NODE(p, Expression);
51 p->op = OP_ADD;
52 p->ty = expr->ty;
53 p->kids[0] = addr;
54 {
55     union value val;
56     val.i[0] = offset;
57     val.i[1] = 0;
58     p->kids[1] = Constant(addr->coord, T(INT), val);
59 }
60 return p;
61 }

```

图 4.65 CheckInitConstant()

我们可用以下全局变量的初始化来说明图 4.65 第 29 至 60 行的代码。可以发现，在汇编代码中，符号 `number` 和 `arr` 实际上对应一个地址常量。在 C 语言中，`&number` 的类型是指针类型，但在汇编指令中，我们使用符号 `number` 即可表示一个地址常量，图 4.65 第 29 至 33 行对此进行预处理。第 34 至 45 行用于处理形如 `arr[2]` 或 `a.b.c` 这样的地址常量，它们的共同特征是最终化简为形如 `addr+k` 的形式，其中 `addr` 为类似 `number` 或 `arr` 的地址常量，而 `k` 为把各偏移进行累加所得到的常量。对于对下 `arr[2]` 来说，按数组的寻址方式，`arr[2]` 对应的地址为 `arr+2*sizeof(int)`，即 `arr+40`。第 50 至 59 行的代码用于为 `addr+k` 构造一个进行 `addr` 和 `k` 之间加法运算的语法树结点。第 47 行的 if 条件要求 `addr` 是标识符，在此前提下，`addr` 可以是数组名或者函数名，当然若 `addr` 为标识符且“第 14 行的形参 `expr` 是形如`&number` 的表达式”也是可以的。

```

int number;
int arr[4][5];
int * ptr1 = &number;
int * ptr2 = arr[2];
//////////////对应的汇编/////////////
ptr1: .long    number
ptr2: .long    arr + 40

```

#### 4.4.4 内部连接和外部连接

在这一小节中，我们来讨论一下 C 语言的内部连接(Internal Linkage)和外部连接(External Linkage)。C 标准 ansi.c.txt 中“连接 (Linkage)”的规定是：出现在不同作用域或者相同作用域的多个重名声明，它们可代表内存中的同一个对象，而把多个重名声明对应到同一内存对象的过程称为连接 (Linkage)，可分有内部连接 (Internal Linkage)、外部连接 (External Linkage) 和无连接 (None of Linkage)。此处我们讨论的连接主要是针对重名的多个外部声明而言，而不是“连接器 (Linker) 所做的把多个目标文件连接 (Linking) 成一个可执行程序”的工作。在英文的语境中，这两者分别对应的单词为 `Linkage` 和 `Linking`，但在中文的语境中，都被译为“连接”。可能我们博大精深的汉语，在表达诗词歌赋方面也许天下无敌，但在描述科技术语时有时会略感乏力。其中的部分原因，或许源于计算机相关学科的很多术语本身就没有正式定义，比如热了好几年的“云计算”，但要回答“什么是 Cloud Computing”还真不是件容易的事情。而在亚洲热播的《甄嬛传》，传到美国时，老美的翻译在碰到名句

“贱人就是矫情”时，也只好放弃。

按 C 标准的规定，在 C 文件中第一次出现的外部声明中，若有关键字 static 则相应标识符为内部连接，而在“有 extern 关键字”或者“即无 static 也无 extern”时为外部连接。C 标准还根据是否有初值，把外部声明分为带初值的定义（Definition）和不带初值的“尝试性定义（Tentative Definition）”，尝试性定义还要求没有 extern 关键字，即不是形如“extern int a;”的声明。如果在 C 文件中出现了定义，则与其同名的尝试性定义就只被当作冗余的声明；如果没有定义，则同名的多个尝试性定义会被合并成一个初值为 0 的定义。

C 标准引入内部连接、外部连接、定义和尝试性定义等概念的目的，主要是为了允许存在重名的多个外部声明。外部连接对应的是“在多个 C 文件中出现的同名且类型相容的外部声明”，这些声明指向全局数据区中的同一个对象；而内部连接对应的是“在同一个 C 文件中出现的同名且类型相容的外部声明”，这些声明指向静态数据区中的同一个对象。一个进程的地址空间可分为：代码区、栈区、堆区和全局静态数据区。如果分得更细一点，我们还可把全局静态数据区分为全局数据区（存放全局变量），静态数据区（存放 static 变量）和常量区（存放浮点数和字符串等常量）。

需要注意的是，此处讲的外部声明是从 C 文法中的非终结符 external-declaration 的角度出发，指的是在函数体外出现的声明，并不是“带关键字 extern 的声明”。由于在函数体内出现的局部变量和形参，且其存储空间是在栈中动态分配，并不在全局静态数据区中，所以 C 标准规定这些标识符为“无连接（None of linkage）”。这意味着在函数体内的同一作用域中，不允许存在重名的局部声明。对于内部连接来说，由于仅涉及到同一个 C 文件内的同名外部声明，C 编译器本身就能处理；而对于外部连接而言，由于涉及到了多个 C 文件中的同名外部声明，最终需要由连接器来处理。

这些规则有点晦涩，下面，我们举一个例子来说明，如图 4.66 所示。第 2 行的 ok1 为内部连接且是尝试性定义，而第 4 行的 extern int ok1 由于不是对 ok1 的第一次声明，其中的 extern 只表示第 4 行是个声明，既不是定义也不是尝试性定义，所以第 2 行和第 4 行之间没有矛盾。第 6 行和第 8 行的 ok2 由于不带 extern 或 static 关键字，被当作外部连接且为尝试性定义，第 10 行的 ok2 带有初值，为定义（Definition），此时第 6 行和第 8 行的 ok2 被当作冗余的声明。第 12 行和第 14 行的 ok3 都为内部连接且为尝试性定义，不带有初值，最终会对应同一个 static int 对象。

```

1 // internal linkage, tentative definition
2 static int ok1;
3 // declaration of ok1
4 extern int ok1;
5 // external linkage,tentative definition
6 int ok2;
7 // external linkage,tentative definition
8 int ok2;
9 // external linkage,definition
10 int ok2 = 2;
11 // internal linkage,tentative definition
12 static int ok3;
13 // internal linkage,tentative definition
14 static int ok3;
15 // external linkage, declaration of err1
16 extern int err1;
17 // internal linkage,tentative definition
18 static int err1;
19 // external linkage,tentative definition
20 int err2;

```

```

21 // internal linkage,tentative definition
22 static int err2;
23 // internal linkage,tentative definition
24 static int err3;
25 // external linkage,tentative definition
26 int err3;
27 // external linkage,definition
28 int err4 = 0;
29 // external linkage,definition
30 int err4 = 0;
31 // external linkage,tentative definition
32 int err5;
33 // external linkage,tentative definition
34 double err5;
35 void f(void){
36 //declaration of ok3
37 extern int ok3;
38 //declaration of ok4,external linkage
39 extern int ok4;
40 // no linkage, definition
41 int ok5;
42 }

```

图 4.66 Linkage

在图 4.66 第 16 行的 `extern int err1` 只是对 `err1` 的声明,但由于是对 `err1` 的第一次声明,所以关键字 `extern` 还把 `err1` 置为外部连接,而第 18 行的声明则试图把 `err1` 设置为内部连接,这就产生了矛盾, C 标准中称这种冲突的后果为“未定义的 (Undefined)”, GCC 和 Clang 编译器视这种矛盾为错误。

而对于图 4.66 第 20 行中既没有 `extern` 也没有 `static` 的外部声明来说, `err2` 为外部连接且为尝试性定义,这与第 22 行发生矛盾。第 24 行和第 26 行的 `err3` 也有类似矛盾。第 28 行和第 30 行都是对 `err4` 的定义,而定义只能有一个。第 32 行和第 34 行则企图把 `err5` 声明为不相容的类型,虽然 `int` 型和 `double` 型的变量可相互赋值,但 `int` 型和 `double` 型并不是相容类型。相容类型意味着内存中的数据不需要进行格式转换,就可以看成与之相容的另一种类型。而 `int` 型和 `double` 型之间的数据格式是不一样的,需要经过转换。第 37 行的 `ok3` 出现在函数体内,且带了 `extern` 关键字,由于我们在第 12 行已经把 `ok3` 设置为内部连接,所以第 37 行的 `ok3` 仍然是内部连接,且只是一个声明。而由于之前没有对 `ok4` 的外部声明,第 39 行的声明会把 `ok4` 设为外部连接。第 41 行的 `ok5` 只是不带初值的局部变量,被视为“无连接”,即不允许在同一作用域中出现同名的 `ok5`。

有了这几小节的基础,我们再去看 `declchk.c` 中的函数 `CheckGlobalDeclaration()`、`CheckFunction()` 和 `CheckLocalDeclaration()` 就会轻松了许多。

#### 4.4.5 外部声明的语义检查

在前面几小节的基础上,我们基本上已经把球从后场带到对方球门前了,就差临门一脚了。在这一节中,我们来分析一下对全局变量进行语义检查的函数 `CheckGlobalDeclaration()`,和对函数定义进行语义检查的 `CheckFunction()`。对全局变量进行检查的主要代码如图 4.67 所示,我们省略了一些细节。图 4.67 第 7 行的 `CheckDeclarationSpecifiers()`、第 22 行的 `CheckDeclarator()` 和第 24 行的 `DeriveType()` 正是我们构建类型结构的三部曲,第 31 行调用 `LookupID()` 函数查询符号表,如果是第一次遇到此全局变量,就在第 32 行调用 `AddVariable()`

将其加入到符号表中。如果是带初值的全局变量，则在第 36 行调用 CheckInitializer() 函数来对初值进行检查；由于全局变量的初值必须是常量，我们还需要在第 37 行调用 CheckInitConstant() 函数来处理。第 39 至 44 行的代码用于处理我们在上一节介绍的“内部连接”和“外部连接”，sclass 为 0 表示外部声明中没有 static 也没有 extern，sclass 为 TK\_EXTERN 表示带有 extern，sclass 为 TK\_STATIC 则表示带有 static 关键字。由于可能存在多个重名的外部声明，我们会在第 46 行调用 IsCompatibleType() 来检查一下重名声明的类型是否相容，在相容的情况下，会在第 50 行调用 CompositeType() 函数进行类型合成。第 52 至 59 行用于检查全局变量是否重复定义，第 56 行在全局变量对应的符号中保存“由若干个 struct initData 对象构成的链表”，其中包含了用于初始化全局变量的各个初值。图 4.67 中所涉及到的各个函数，我们都已在前文分析过。这是我们在本节中的第一脚射门，在中场和后场组织好的前提下，前锋这临门一脚就轻松多了。

```

1 static void CheckGlobalDeclaration(
2             AstDeclaration decl){
3     AstInitDeclarator initDec;
4     Type ty;
5     Symbol sym;
6     int sclass;
7     CheckDeclarationSpecifiers(decl->specs);
8     // typedef int INT_ARR[4];
9     if (decl->specs->sclass == TK_TYPEDEF) {
10         CheckTypedef(decl);
11         return;
12     }
13     ty = decl->specs->ty;
14     sclass = decl->specs->sclass;
15     if (sclass == TK_REGISTER || sclass == TK_AUTO) {
16         Error(&decl->coord, "Invalid storage class");
17         sclass = TK_EXTERN;
18     }
19     // check init-declarator list.
20     initDec = (AstInitDeclarator)decl->initDecls;
21     while (initDec) {
22         CheckDeclarator(initDec->dec);
23         .....
24         ty = DeriveType(initDec->dec->tyDrvList,
25                         decl->specs->ty,&initDec->coord);
26         if (ty == NULL) {
27             Error(&initDec->coord, "Illegal type");
28             ty = T(INT);
29         }
30         .....
31         if ((sym = LookupID(initDec->dec->id)) == NULL) {
32             sym = AddVariable(initDec->dec->id,
33                               ty, sclass,&initDec->coord);
34         }
35         if (initDec->init) {
36             CheckInitializer(initDec->init, ty);
37             CheckInitConstant(initDec->init);
38         }
39         sclass =
40             (sclass == TK_EXTERN) ? sym->sclass : sclass;

```

```

41     if ((sclass == 0 && sym->sclass == TK_STATIC)
42         || (sclass != 0 && sym->sclass != sclass)) {
43         Error(&decl->coord, "Conflict linkage ...");
44     }
45     .....
46     if (! IsCompatibleType(ty, sym->ty)){ //
47         Error(&decl->coord, "Incompatible ...");
48         goto next;
49     }else{
50         sym->ty = CompositeType(sym->ty, ty);
51     }
52     if(initDec->init){
53         if (sym->defined) {
54             Error(&initDec->coord, "redefinition ...");
55         }else{
56             AsVar(sym)->idata = initDec->init->idata;
57             sym->defined = 1;
58         }
59     }
60 next:
61     initDec = (AstInitDeclarator) initDec->next;
62 }
63 }
```

图 4.67 CheckGlobalDeclaration()

另一脚射门是关于函数定义的，如图 4.68 所示。在第 10 行、第 14 行和第 16 行调用的函数仍然是构建类型结构的三部曲，第 22 行查符号表，如果是第一次遇到此函数，则在第 25 行调用 AddFunction() 来把函数相关信息加入到符号表；如果之前已把此标识符声明为其他类型，则在第 27 行报错。如果之前已经对此函数名作过声明，则在第 35 行调用函数 IsCompatibleType() 来检查一下两次声明的类型是否相容。如果相容就在第 39 行调用 CompositeType() 函数进行类型合成。第 46 行创建的向量 loops 存放循环语句，第 47 行的向量 breakable 用于存放循环语句和 switch 语句，而第 48 行的向量 switches 用于存放 switch 语句，我们在对语句进行语义检查时使用过这些向量。由于函数的形参和局部变量要处在同一个作用域中，第 51 行调用的 RestoreParameterListTable() 函数用于恢复“我们在第 14 行调用 CheckDeclarator() 函数对形参的类型进行检查时，通过 SaveParameterListTable() 函数所保存的符号表”，第 53 至 57 行将各个形参数加入到此符号表中。而第 61 行调用的 CheckCompoundStatement() 则用于对函数体进行语义检查，C 语言的函数体实际上就是一个复合语句。之后在第 62 行调用 ExitScope() 退出当前作用域。

```

1 void CheckFunction(AstFunction func) {
2     Symbol sym;
3     Type ty;
4     int sclass;
5     Label label;
6     AstNode p;
7     Vector params;
8     Parameter param;
9     func->fdec->partOfDef = 1;
10    CheckDeclarationSpecifiers(func->specs);
11    if ((sclass = func->specs->sclass) == 0){
12        sclass = TK_EXTERN;
13    }
```

```

14 CheckDeclarator(func->dec);
15 ....
16 ty = DeriveType(func->dec->tyDrvList,
17                   func->specs->ty,&func->coord);
18 if (ty == NULL) {
19     Error(&func->coord, "Illegal function type");
20     ty = DefaultFunctionType;
21 }
22 sym = LookupID(func->dec->id,!ADD_PLACEHOLDER);
23 if (sym == NULL){
24     func->fsym = (FunctionSymbol)
25         AddFunction(func->dec->id, ty, sclass,&func->coord);
26 } else if (sym->ty->categ != FUNCTION){
27     Error(&func->coord, "Redeclaration as a function");
28     func->fsym = (FunctionSymbol)
29         AddFunction(func->dec->id, ty, sclass,&func->coord);
30 }else{
31     func->fsym = (FunctionSymbol)sym;
32     if (sym->sclass == TK_EXTERN && sclass == TK_STATIC){
33         Error(&func->coord, "Conflict function linkage");
34     }
35     if (! IsCompatibleType(ty, sym->ty)){
36         Error(&func->coord, "Incompatible ...");
37         sym->ty = ty;
38     }else{
39         sym->ty = CompositeType(ty, sym->ty);
40     }
41     if (func->fsym->defined){
42         Error(&func->coord, "Function redefinition");
43     }
44 }
45 func->fsym->defined = 1;
46 func->loops = CreateVector(4);
47 func->breakable = CreateVector(4);
48 func->swtches = CreateVector(4);
49 CURRENTF = func;
50 FSYM = func->fsym;
51 RestoreParameterListTable();
52 {
53     Vector v= ((FunctionType)ty)->sig->params;
54     FOR_EACH_ITEM(Parameter, param, v)
55         AddVariable(param->id, param->ty,
56                     param->reg ? TK_REGISTER : TK_AUTO,&func->coord);
57     ENDFOR
58     FSYM->locals = NULL;
59     FSYM->lastv = &FSYM->locals;
60 }
61 CheckCompoundStatement(func->stmt);
62 ExitScope();
63 ....
64 }

```

图 4.68 CheckFunction()

而用于对局部变量声明进行语义检查的函数 CheckLocalDeclaration(), 与图 4.67 中的

`CheckGlobalDeclaration()`有很多相似的地方，较大的不同在于局部变量的初值不一定非得是常量，因此我们在调用 `CheckInitializer()` 检查完初值后，不再需要调用 `CheckInitConstant()`。只有对函数体内出现的 `static` 变量，我们才需要调用 `CheckInitConstant()` 来检查其初值是否为常量。另一个较大的区别是，函数体中出现的局部变量是“无连接（No Linkage）”，同一作用域中不可以出现同名的局部变量，因此也就不需要调用 `CompositeType()` 函数进行相容类型的类型合成。

至此，我们完成了语义检查，UCC 编译器可把语义检查后得到的语法树打印出来，这个工作由 `ucl\dumpast.c` 中的 `DumpTranslationUnit()` 函数来完成，相关代码并不太复杂，这里从略。在下一章中，我们将讨论中间代码生成。

## 4.5 本章习题

1. 在语义检查时，UCC 编译器是如何处理以下数组名的。

```
int number[16];
void f(int arr[1000]){
    sizeof(number);
    number+1;
    &number+1;
}
```

2. 请用 UCC 编译并运行以下程序，观察其运行结果。之后，在 UCC 编译器的源代码中，删去图 4.17 第 10 至 16 行所示的代码，重新生成 UCC 编译器后，再用新生成的 UCC 编译并运行以下程序，观察其运行结果，并解释其原因。

```
#include <stdio.h>
short s = -1;
int b;
int main(int argc,char * argv[]){
    b = (int)((unsigned int)s) >> 1;
    printf("%d \n",b);
    return 0;
}
```

3. 按 C 标准的要求，`break` 语句必须出现在 `switch` 或循环语句中，UCC 编译器是如何对此进行检查的。

# 第 5 章 中间代码生成及优化

## 5.1 中间代码生成简介

在语法分析和语义检查阶段，我们始终在与语句、表达式和外部声明这 3 个概念打交道。通过声明我们最终建立起了相应的类型结构，并在符号表中保存了相关标识符的类型信息，到了中间代码生成阶段，我们就不再需要处理外部声明了，只需要为语句和表达式生成相应的中间代码即可。文件 ucltranexpr.c 用于为表达式生成中间代码，文件名 tranexpr 是 Translate Expression 的缩写，而 ucltranstmt.c 则用于为语句生成中间代码。我们先结合图 5.1 所示的例子来讨论。图 5.1 第 1 至 9 行的递归函数用于计算  $n$  阶乘，第 12 至 15 行的 while 循环用于打印出  $1!$  至  $10!$  的值，第 19 至 45 是“UCC 编译器所生成的中间代码”的字符串形式，UCC 编译器内部会用三地址码的结构来表示中间代码。“龙书”把语法树和三地址码都视为中间代码，这里我们在讨论 UCC 编译器时，如果不作特别声明，中间代码指的是三地址码。三地址码包含两个源操作数、一个目的操作数及一个运算符。例如对于“ $t1 : a+b$ ”而言，加号“+”是运算符， $a$  和  $b$  是两个源操作数，而  $t1$  是目的操作数（即用于存放运算结果），这三个操作数对应三个“地址”，所以称这样的中间代码为三地址码。

```

1 // hello.c                                24 //
2 #include <stdio.h>                         25 BB3:
3 int f(int n){                                26   t0 : n + -1;
4   if(n < 1){                                27   t1 : f(t0);
5     return 1;                                28   t2 : n * t1;
6   }else{                                    29   return t2;
7     return n *f(n-1);                        30 BB4:
8   }                                         31   ret
9 }                                         32 /////////////////////////////////
10 int main(int argc,char * argv[]){           33 function main
11   int i = 1;                                34   i = 1;
12   while(i <= 10){                           35   goto BB7;
13     printf("f(%d)= %d\n",i,f(i));          36 BB6:
14     i++;                                 37   t0 :&str0;
15   }                                         38   t1 : f(i);
16   return 0;                                39   printf(t0, i, t1);
17 }                                         40   ++i;
18 // hello.uil                               41 BB7:
19 function f                                42   if (i <= 10) goto BB6;
20   if (n >= 1) goto BB3;                     43 BB8:
21 BB1:                                     44   return 0;
22   return 1;                                45   ret
23   goto BB4;

```

图 5.1 基本块

图 5.11 第 20 行的中间代码表示有条件的跳转，第 23 行的 goto 表示无条件的跳转，在汇编语言中，都有与之对应的汇编指令。低级语言中的“有条件的跳转”会引起控制流的转移，由此可以实现高级语言中的分支语句 if 和循环语句 while 等控制结构。当然，函数返回也是一种控制流的变化，在 UCC 编译器生成的中间代码中，第 22 行的 return 1 只是把返回值设为 1，真正的返回动作由第 31 行的 ret 指令来完成。这样处理的好处在于，即

便在 C 语言中出现了多条 return 语句（如第 5 行和第 7 行的 return 语句），但在中间代码层次，我们只在第 22 行设置返回值为 1，第 29 行设置返回值为 t2，真正的返回动作只需要在同一个地方处理（即第 30 行的 ret 指令），这也意味着我们从函数的入口处开始执行，离开函数时也只有一个出口。执行“函数调用”时，我们要依次把实参和返回地址入栈，之后无条件跳转到函数起始地址，因此函数调用也可看成是一种无条件跳转。生成中间代码后，我们还需要进行代码优化，此时我们需要考虑控制流的变化。我们期望把相邻的若干条中间代码当作一个整体来处理，例如第 26 至 29 行的这 4 条中间代码，控制流只能从这个整体的第一条指令进入（此处即第 26 行对应的中间代码），离开时从基本块中的最后一条指令离开（此处即第 29 行的中间代码），我们称这样的“整体”为一个基本块（Basic Block），第 25 行的“BB3”表示第 3 个基本块。由此，整个 C 程序就可由若干个基本块构成。

对图 5.1 第 19 至 45 行的中间代码而言，从静态来看，这些基本块是从上至下依次排列的，我们用一个链表就可以存放这些基本块。但从条件跳转和无条件跳转的动态语义来看，若把第 41 行的基本块 BB7 看成一个结点，则执行完第 42 行的条件跳转语句后，接下来我们可能执行的是第 44 行的中间代码，也可能是第 37 行的中间代码。这就相当于存在一条由第 41 行的基本块 BB7 指向第 43 行的基本块 BB8 的有向边，同时从 BB7 到 BB6 也存在一条有向边。由此，基本块成了点，而“控制流的转移”就成了有向边，可把整个程序的动态执行的路线看成数据结构中的“图”，这个由基本块构成的图被称为控制流图（Control Flow Graph），简写为 CFG。后续的分析和优化都是基于“控制流图”这样的数据结构进行。需要说明的是，UCC 编译器的中间代码优化工作只在基本块中进行，并没有进行函数间（过程间）的代码优化，因此 UCC 编译器在划分基本块时，并没有把函数调用当作基本块的最后一条指令，例如在图 5.1 第 36 行的基本块 BB6 中，第 39 行调用的函数 printf() 并不是基本块的最后一条指令。在中间代码优化阶段，ucl\simp.c 中的 PeepHole() 函数会对 CALL 指令进行优化处理，PeepHole 译为“窥孔优化”，“窥孔”的含义是我们通过一个小孔或小窗口来看世界，一次只看到世界的一小部分（局部）。在 UCC 编译器中，这个孔的大小一般就限定在一个“基本块”内。例如，对于以下两条中间代码来说：

```
t1 = f();
num = t1;
```

由于 UCC 编译器没有把函数调用 f() 置于基本块的最末尾，因此这 2 条中间代码就可以处于同一个基本块中，在 PeepHole() 函数对其所处基本块进行“窥孔优化”时，就会发现临时变量 t1 是多余的，这 2 条中间代码可优化为如下所示的中间代码：

```
num = f();
```

为了方便进行这样的优化，UCC 编译器在划分基本块时，并没有把函数调用当作控制流的转移。但我们知道，在真正执行机器指令时，函数调用确实会导致控制流的转移，UCC 编译器在后续阶段产生 x86 汇编指令时，还会在 ucl\x86.c 的 EmitBlock() 函数中对 CALL 指令进行特殊判断，从而保存函数调用前用到的一些寄存器。

简单来说，为了存放如图 5.1 第 19 至 45 行的中间代码，我们需要由若干个基本块构成的链表结构；为了描述控制流在基本块间的动态转移，我们需要“控制流图”的数据结构。图 5.2 第 2 至 8 行的结构体 irinst 用于描述一条“三地址码”中间代码，第 7 行的 opds[3] 用于存放 3 个操作数，每个操作数由一个符号对象 struct symbol 来表示，我们在第 2.5 节时简介过符号的相关概念。第 6 行的 opcode 用于存放运算符。由于一个基本块可以包含若干条的中间代码，我们可用第 3 行的 prev 和第 4 行的 next 来构成双向链表结构，第 5 行的 ty 用于记录运算结果的类型信息。

```
1 // Intermediate Representation
2 typedef struct irinst{
3     struct irinst *prev;
```

```
4 struct irinst *next;
5 Type ty;
6 int opcode;
7 Symbol opds[3];
8 } *IRInst;
9 // control flow graph edge
10 typedef struct cfgedge{
11 BBlock bb;
12 struct cfgedge *next;
13 } *CFGEdge;
14 // Basic Block
15 struct bblock{
16 struct bblock *prev;
17 struct bblock *next;
18 Symbol sym;
19 // successors
20 CFGEdge succs;
21 // predecessors
22 CFGEdge preds;
23 struct irinst insth;
24 // number of instructions
25 int ninst;
26 // number of successors
27 int nsucc;
28 // number of predecessors
29 int npred;
30 // number of references
31 int ref;
32 };
33 static void AddPredecessor(BBlock bb, BBlock p) {
34 CFGEdge e;
35 ALLOC(e);
36 e->bb = p;
37 e->next = bb->preds;
38 bb->preds = e;
39 bb->npred++;
40 }
41 static void AddSuccessor(BBlock bb, BBlock s){
42 CFGEdge e;
43 ALLOC(e);
44 e->bb = s;
45 e->next = bb->succs;
46 bb->succs = e;
47 bb->nsucc++;
48 }
49 void DrawCFGEdge(BBlock head, BBlock tail){
50 AddSuccessor(head, tail);
51 AddPredecessor(tail, head);
52 }
53
54 void AppendInst(IRInst inst){
55 assert(CurrentBB != NULL);
56 CurrentBB->insth.prev->next = inst;
```

```

57 inst->prev = CurrentBB->insth.prev;
58 inst->next = &CurrentBB->insth;
59 CurrentBB->insth.prev = inst;
60 CurrentBB->ninst++;
61 }

```

图 5.2 与中间代码有关的数据结构

图 5.2 第 15 至 32 行的结构体用于刻画一个基本块，第 16 行的 `prev` 和第 17 行的 `next` 用于构造由若干个基本块形成的双向链表。双向链表描述了如图 5.4 所示的基本块的“静态结构”；而在动态结构“控制流图”中，一个结点可以有多个前驱，也可以有多个后继，第 20 行的 `succs` 用于记录当前基本块的所有后继结点，而第 22 行的 `preds` 用于记录其所有的前驱结点。第 25 行的 `ninst` 用于记录当前基本块中有多少条中间代码，第 27 行的 `nsucc` 用于记录后继结点的个数，而 29 行的 `npred` 则用于记录前驱结点的个数。第 18 行的 `sym` 用于存放基本块的名称，例如“BB3”等。第 23 行的 `insth` 对应一条占位用的中间代码，仅仅用于充当双向链表的头结点，并不对应任何实际的代码。

由于一个基本块 `Bx` 可以有多个前驱 {`B1, B2, B3, ..., Bn`}，每个前驱到基本块 `Bx` 都存在一条有向边。同理，该基本块 `Bx` 也可以有多个后继，从 `Bx` 到各个后继结点也存在相应的有向边。图 5.2 第 10 至 13 行的结构体 `struct cfgedge` 用于描述一条有向边，第 11 行的 `bb` 域用于存放一条有向边的前驱（或者后继）；第 12 行的 `next` 域用于构成若干个前驱（或者后继）形成的单向链表。第 49 行的函数 `DrawCFGEdge(head,tail)` 用于构造一条从基本块 `head` 指向基本块 `tail` 的有向边，这意味着 `tail` 是 `head` 的后继结点，我们通过第 50 行的函数 `AddSuccessor()` 把 `tail` 加入到基本块 `head` 的 `succs` 域所指向的后继链表中；同时，`head` 是 `tail` 结点的前驱，UCC 把 `head` 加入到基本块 `tail` 的 `preds` 域所指向的前驱链表中，这个工作由第 51 行的函数 `AddPredecessor()` 来实现。

```
head --> tail
```

图 5.2 第 54 行的函数 `AppendInst()` 用于往当前基本块中添加一条中间代码，第 55 至 59 行的 4 条语句用于实现双向链表的插入操作。

接下来，我们来初步了解一下中间代码生成的总体执行过程，如图 5.3 所示，第 55 至 64 行的 `Translate()` 函数实现了由抽象语法树到三地址码的翻译，第 57 行的 `while` 循环依次对当前 C 文件中的各个函数进行翻译，实际的工作由第 60 行的 `TranslateFunction()` 来完成。图 5.3 第 38 至 54 行给出了函数 `TranslateFunction()` 的主要代码，按照我们前面对 `return` 语句的处理，不论函数内部的控制流有多复杂，整个函数定义都只有一个入口和一个出口。第 43 行和第 44 行分别调用 `CreateBBlock()` 来创建这两个基本块，第 20 至 26 行给出了 `CreateBBlock()` 函数的代码，第 23 行用于设置头结点为空指令 `NOP`（`No Operation` 的缩写，即“无任何实际运算”）。第 46 行调用 `TranslateStatement()` 实现了对函数体的翻译，函数体实际上是一个复合语句。第 1 至 16 行列出了一个函数指针表，第 2 至 15 行的各函数完成了各语句的翻译工作，第 17 至 19 行的 `TranslateStatement()` 函数只是查询这个函数指针表而已。我们会在后续章节对第 2 至 15 行中的各个函数进行分析。第 48 行的 `Optimize()` 函数用于对生成的中间代码进行优化，第 50 行的 `while` 循环用于给各个基本块命名，形如“BB1”和“BB2”等。图 5.3 第 38 至 54 行就是中间代码生成及优化的主要流程。

```

1 static void (* StmtTrans[])(AstStatement) = {
2     TranslateExpressionStatement,
3     TranslateLabelStatement,
4     TranslateCaseStatement,
5     TranslateDefaultStatement,
6     TranslateIfStatement,
7     TranslateSwitchStatement,
8     TranslateWhileStatement,

```

```
9   TranslateDoStatement,
10  TranslateForStatement,
11  TranslateGotoStatement,
12  TranslateBreakStatement,
13  TranslateContinueStatement,
14  TranslateReturnStatement,
15  TranslateCompoundStatement
16  };
17 static void TranslateStatement(AstStatement stmt){
18  (* StmtTrans[stmt->kind - NK_ExpressionStatement])(stmt);
19 }
20 BBlock CreateBBlock(void) {
21  BBlock bb;
22  CALLOC(bb);
23  bb->insth.opcode = NOP;
24  bb->insth.prev = bb->insth.next = &bb->insth;
25  return bb;
26 }
27 void StartBBlock(BBlock bb) {
28  IRInst lasti;
29  CurrentBB->next = bb;
30  bb->prev = CurrentBB;
31  lasti = CurrentBB->insth.prev;
32  if (lasti->opcode != JMP
33      && lasti->opcode != IJMP) {
34      DrawCFGEdge(CurrentBB, bb);
35  }
36  CurrentBB = bb;
37 }
38 static void TranslateFunction(AstFunction func) {
39  BBlock bb;
40  FSYM = func->fsym;
41  ...
42  TempNum = 0;
43  FSYM->entryBB = CreateBBlock();
44  FSYM->exitBB = CreateBBlock();
45  CurrentBB = FSYM->entryBB;
46  TranslateStatement(func->stmt);
47  StartBBlock(FSYM->exitBB);
48  Optimize(FSYM);
49  bb = FSYM->entryBB;
50  while (bb != NULL){
51      bb->sym = CreateLabel();
52      bb = bb->next;
53  }
54 }
55 void Translate(AstTranslationUnit transUnit){
56  AstNode p = transUnit->extDecls;
57  while (p){
58      if (p->kind == NK_Function
59          && ((AstFunction)p)->stmt) {
60          TranslateFunction((AstFunction)p);
61      }
62 }
```

```

62     p = p->next;
63 }
64 ...
65 }

```

图 5.3 Translate()

需要注意的是图 5.3 第 43 行调用的 CreateBBBlock() 函数返回后，新创建的基本块对象并未加入到双向链表中。只有调用了第 27 行的 StartBBBlock(bb) 函数后，我们才会把函数参数 bb 所指向的基本块对象加入到双向链表中，第 29 至 30 行的代码完成了这个插入操作。图 5.3 第 32 行的 if 语句用于检查“当前基本块的最后一条指令是否会使控制流转移到参数 bb 所指向的基本块”，如果可能出现这样的转移，我们就会在第 34 行调用 DrawCFGEdge() 函数，构造一条由当前基本块指向参数 bb 基本块的有向边，之后在第 36 行使参数 bb 成为新的当前基本块。基本块末尾的无条件跳转指令 JMP，会导致控制流转移到不相邻的基本块上，例如图 5.1 基本块 BB1 第 23 行的 goto BB4 指令，就会跳往与 BB1 不相邻的基本块 BB4。对于这种情况，在生成无条件跳转指令时，UCC 会进行有向边的构造，相应的操作可参见 uclgen.c 中的函数 GenerateJump()。类似的，如果基本块的最末尾一条指令是如下所示的间接跳转，我们也不调用图 5.3 第 34 行的 DrawCFGEdge() 函数，而是由 uclgen.c 中的函数 GenerateIndirectJump() 根据实际的跳转目标来构造有向边。这两个函数并不复杂，这里从略。在进行 switch 语句的翻译时，我们会用到间接跳转。

```

//根据 t0 的值来确定跳转的目标
goto (BB1, BB2, BB3, )[t0];

```

而如果当前基本块的最末一条指令是“有条件跳转指令”，则控制流还是可能流向相邻的下一个基本块的，此时我们需要调用图 5.3 第 34 行 DrawCFGEdge() 函数来构造有向边，例如图 5.1 的第 42 行。当然，如果最末一条指令不是跳转指令，那控制流一定是流向相邻的下一个基本块，此时我们也需要调用图 5.3 第 34 行的 DrawCFGEdge() 函数，例如图 5.1 的第 40 行。对于图 5.1 中的 main() 函数而言，图 5.4 给出了其基本块的静态结构和动态结构，静态结构采用的是双向链表，而动态结构采用的是控制流图。

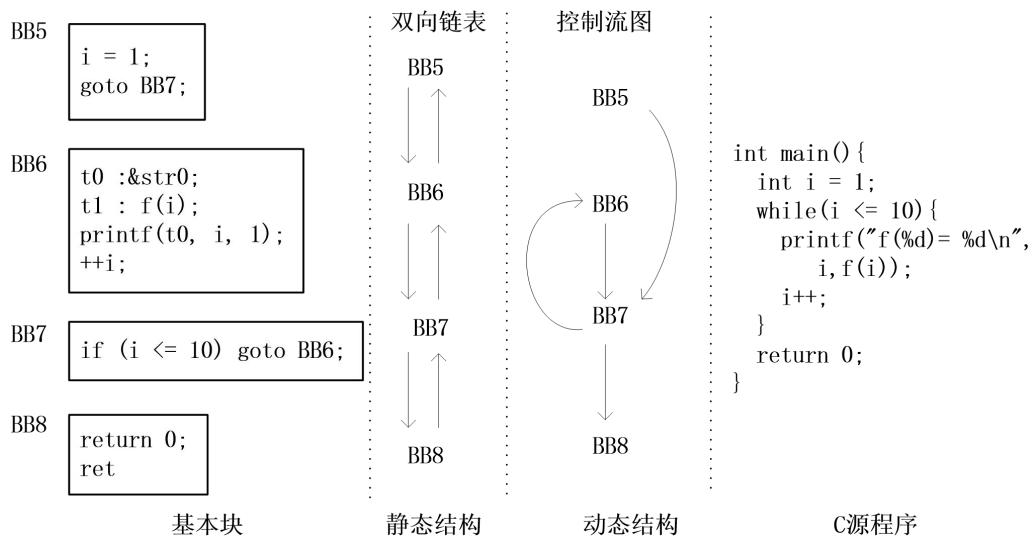


图 5.4 基本块的静态结构和动态结构

## 5.2 表达式的翻译

## 5.2.1 布尔表达式的翻译

我们仍然按照语法分析和语义检查时的思路，先讨论表达式的翻译，再处理语句。表达式从概念上来说，可分为算术表达式和布尔表达式，在一些编程语言（例如 Java）中对这两者是有严格区分的，算术表达式的结果是整数或浮点数等，而布尔表达式的结果是逻辑上的真或假。布尔是英国数学家，由于布尔较早进行了关于“与或非”逻辑运算的研究，为了纪念这位先驱，在 Java 中引入了关键字 boolean，而在 C++ 中引入了 bool 关键字来表达逻辑上的真或假。C 语言中并没有专门的布尔类型，而是把非 0 的值当作真，把 0 当作假。

```
int a, b;
if(a+b) { // Java 中非法, a+b 为算术值, 非布尔值; c 中合法
}
if(a == b) { // a == b 为布尔值, 在 C 或 Java 中都合法
}
```

为了讨论的方便，我们不妨认为 C 语言存在布尔表达式，例如  $a \&& b$ 、 $a|b$  或者  $a>b$  等，也存在算术表达式，例如  $a+b$  或者  $a \& b$ ，但是不存在 C 程序员可见的布尔类型关键字 bool 或者 boolean。对于算术表达式而言，C 编译器需要对整个表达式进行求值，例如，我们要计算出“a 和 b 相加的值”，或者“a 和 b 按位与的值”，并把运算结果保存到一个临时变量中，这个临时变量的值就代表了整个表达式的值。而对于布尔表达式来说，我们并不会对整个表达式进行求值，而是生成相应的跳转指令。对于  $a \&& b$  而言，当 a 为假时，不论 b 为何值， $a \&& b$  的值都为假，此时我们并不需要对 b 进行求值。若把“ $a \&& b$ ”换成“ $f() \&& g()$ ”，则当  $f()$  的返回值为假时，我们不需要对函数  $g()$  进行求值，此时函数  $g()$  根本就没有被调用，这就是 C 语言中的短路运算。

我们先举个简单的例子来说明这些概念，如图 5.5 所示，第 1 至 21 行给出了一个简单的 C 程序，第 22 至 60 行是与之对应的中间代码。对于第 4 至 8 行的 C 代码来说，与其对应的中间代码在第 24 至 31 行，我们需要先对第 4 行的算术表达式  $a+b$  进行求值，第 25 行的中间代码“t0:a+b”表示把“ $a+b$ ”求值后的结果存于临时变量 t0 中，对表达式进行求值的工作由 ucl\tranexpr.c 中的 TranslateExpression() 函数来完成。但对于第 9 行的布尔表达式  $a \&& b$  来说，我们只是生成第 33 行和第 35 行的条件跳转指令，并没有对整个布尔表达式  $a \&& b$  进行求值，当 a 为假时，控制流直接由第 33 行转移到基本块 BB6 处，即不再需要判断 b 的值。产生这些跳转指令的工作由 ucl\tranexpr.c 中的 TranslateBranch() 函数来实现。在 x86 汇编中，跳转指令对应的助记符为 Jump，缩写为 J；但在 Arm 汇编中，跳转指令对应的是 Branch，缩写为 B，单词 Branch 是分支的意思（也有些人写为分枝），这意味着如果是“有条件跳转”的话，控制流就会“开叉”，相当于一根树枝分成若干个细枝了，例如图 5.5 第 35 行的有条件跳转，控制流可能流向第 37 行，也可能流向第 40 行。

```
1 // hello.c
2 int a, b, c;
3 int main(int argc,char * argv[]){
4     if(a+b){
5         c = 1;
6     }else{
7         c = 2;
8     }
9     if(a && b){
10        c = 3;
11    }else{
12        c = 4;
13    }
14     if(a & b){
15         c = 5;
16     }else{
17         c = 6;
18     }
19     c = a || b;
20     return 0;
21 }
22 // hello.uil
23 function main
24 // if(a+b)
25 t0 : a + b;
26 if (! t0) goto BB2;
```

```

27 BB1:
28 c = 1;
29 goto BB3;
30 BB2:
31 c = 2;
32 BB3: // if(a && b)
33 if (!a) goto BB6;
34 BB4:
35 if (!b) goto BB6;
36 BB5:
37 c = 3;
38 goto BB7;
39 BB6:
40 c = 4;
41 BB7: // if(a & b)
42 t1 : a & b;
43 if (!t1) goto BB9;
44 BB8:
45 c = 5;
46 goto BB10;
47 BB9:
48 c = 6;
49 BB10: // c = a || b
50 if (a) goto BB13;
51 BB11:
52 if (b) goto BB13;
53 BB12:
54 c = 0;
55 goto BB14;
56 BB13:
57 c = 1;
58 BB14:
59 return 0;
60 ret

```

图 5.5 布尔表达式和算术表达式

对于图 5.5 第 14 行的算术表达式  $a \& b$  来说，我们需要先对整个表达式  $a \& b$  进行求值，再根据其结果进行跳转，如图第 42 至 43 行所示。比较特殊的是第 19 行的布尔表达式  $a \parallel b$ ，按我们之前的介绍，翻译布尔表达式时，我们只是产生一些跳转指令来改变控制流，如第 50 行和第 52 行所示。但按照第 19 行赋值语句的语义，我们需要在  $a \parallel b$  为真时，给变量  $c$  赋值 1；在  $a \parallel b$  为假时，给变量赋值 0。因此，C 编译器需要产生第 54 至 57 行的代码。这些工作由 ucl\tranexpr.c 中的函数 TranslateBranchExpression() 来完成。

简而言之，如果我们需要显式地求出表达式的值，就调用函数 TranslateExpression()；如果只是需要生成一些跳转指令来改变当前的控制流，则使用 TranslateBranch()。如果我们在调用 TranslateExpression() 来对表达式求值时，遇到的表达式是形如图 5.5 第 19 行的赋值语句中的布尔表达式  $a \parallel b$ ，此时我们就再调用 TranslateBranchExpression() 函数来处理，进而产生如图 5.5 第 50 至 57 行的代码。

趁热打铁，我们现在就来看一下函数 TranslateBranchExpression() 的代码，如图 5.6 第 1 至 18 行所示。第 4 行通过调用 CreateTemp() 函数创建了一个临时变量，不妨设这个临时变量为  $t$ 。我们需要根据布尔表达式的控制流来执行  $t=0$  或者  $t=1$  的代码，第 11 行调用 GenerateMove() 函数产生  $t=0$  的代码，第 15 行则用于产生  $t=1$ 。当表达式 expr 为假时，我们在执行完  $t=0$  后，还要通过第 12 行的 GenerateJump() 函数，产生一条无条件跳转指令，跳过  $t=1$  这个指令。中间代码  $t=0$  是基本块 falseBB 的第一条指令，而  $t=1$  为基本块 trueBB 的第一条指令。在调用第 8 行的 TranslateBranch() 函数来产生跳转指令时，我们需要把 trueBB 和 falseBB 作为跳转目标传递给 TranslateBranch 函数，这样我们才能生成形如图 5.5 第 50 至 52 行的跳转指令。

```

1 static Symbol TranslateBranchExpression(AstExpression expr) {
2   BBlock nextBB, trueBB, falseBB;
3   Symbol t;
4   t = CreateTemp(expr->ty);
5   nextBB = CreateBBlock();
6   trueBB = CreateBBlock();
7   falseBB = CreateBBlock();
8   TranslateBranch(expr, trueBB, falseBB);
9   StartBBlock(falseBB);
10 // t = 0;
11 GenerateMove(expr->ty, t, IntConstant(0));

```

```

12 GenerateJump(nextBB);
13 StartBBlock(trueBB);
14 // t = 1;
15 GenerateMove(expr->ty, t, IntConstant(1));
16 StartBBlock(nextBB);
17 return t;
18 }
19 static Symbol TranslateBinaryExpression(AstExpression expr) {
20 Symbol src1, src2;
21 if (expr->op == OP_OR || expr->op == OP_AND
22     || (expr->op >= OP_EQUAL && expr->op <= OP_LESS_EQ)) {
23     return TranslateBranchExpression(expr);
24 }
25 src1 = TranslateExpression(expr->kids[0]);
26 src2 = TranslateExpression(expr->kids[1]);
27 return Simplify(expr->ty, OPMap[expr->op], src1, src2);
28 }
29 static Symbol (* ExprTrans[]) (AstExpression) = {
30 TranslateCommaExpression,
31 TranslateAssignmentExpression,
32 ...
33 TranslateConditionalExpression,
34 TranslateErrorExpression,
35 TranslateBinaryExpression,
36 ...
37 TranslateUnaryExpression,
38 ...
39 TranslatePostfixExpression,
40 ...
41 TranslatePrimaryExpression,
42 ...
43 TranslateErrorExpression,
44 };
45 Symbol TranslateExpression(AstExpression expr) {
46 return (* ExprTrans[expr->op])(expr);
47 }
48 Symbol Simplify(Type ty, int opcode,
49     Symbol src1, Symbol src2){
50 VariableSymbol t;
51 Symbol p1, p2; int c1, c2;
52 ....
53 switch (opcode) {
54 case LSH:
55 case RSH:// a >> 0 = a << 0 = a
56     if (src2->val.i[0] == 0)
57         return src1;
58     break;
59 ....
60 }
61 add_value:
62 return TryAddValue(ty, opcode, src1, src2);
63 }

```

图 5.6 TranslateBranchExpression()

当我们遇到形如“`c = expression;`”的语句时，由上下文可知，我们需要先对赋值号右侧的表达式进行求值，之后再结果赋给左侧的`c`。对右侧表达式进行求值的运算，由图 5.6 第 45 行的 `TranslateExpression()` 来实现。在第 46 行，我们又遇到了非常熟悉的套路，通过查表来调用相应的处理函数，对应的表格如图第 29 至 44 行所示，分别是第 30 行的逗号表达式、第 31 行的赋值表达式、第 33 行的条件表达式、第 35 行的二元表达式、第 37 行的一元表达式、第 39 行的后缀表达式和第 41 行的基本表达式。我们会在后续章节分别对这些处理函数进行讨论。当我们遇到“`c = a || b`”时，赋值号右侧的“`a || b`”实际上是个布尔表达式，运算符`||`属于二元运算符，通过第 46 行的查表操作，我们会调用第 19 行的函数 `TranslateBinaryExpression()` 来处理。由于当前运算符为`||`，即 `OP_OR`，此时第 21 行的条件会成立，我们就会在第 23 行调用 `TranslateBranchExpression()` 函数来生成以下代码：

```

BB10:    // c = a || b
          if (a) goto BB13;
BB11:
          if (b) goto BB13;
BB12:
          t = 0;
          goto BB14;
BB13:
          t = 1;
BB14:
          c = t;

```

当然，上述最末尾的中间代码“`c = t;`”并不在 `TranslateBranchExpression()` 函数中生成。经过中间代码的优化，我们可以删去多余的临时变量 `t`，就可以得到如图 5.5 第 50 至 57 行的代码。如果想观察未经优化过的中间代码，可把图 5.3 第 48 行的 `Optimize()` 函数调用给注释掉，然后重新 make 编译器 UCC 的源代码。当然，更好的办法是像 GCC 或 Clang 那样，加入一些新的命令行参数，来指定优化的级别。

如果在 TranslateBinaryExpression()中遇到的是形如“ $a << b$ ”或“ $a + b$ ”这样的二元表达式，此时图 5.6 第 21 行的 if 条件就不会成立，我们会调用第 27 行的 Simplify() 函数来做一些简单的优化，例如对于  $a << 0$ ，我们根本就没有必要产生左移指令，因为  $a$  左移 0 位仍然是  $a$  本身。图 5.6 第 48 至 63 行中给出了函数 Simplify() 的部分代码，第 62 行调用的 TryAddValue() 函数主要是用于处理公共子表达式，在第 2 章的图 2.34 中已介绍过这个函数，这里不再重复。

当布尔表达式  $a \parallel b$  出现在  $\text{if}(a \parallel b)$  中时，其翻译的方法就与  $c = a \parallel b$  有所不同，此时只需要产生跳转指令即可，不必生成  $t=0$  或  $t=1$  这样的指令。接下来，我们来分析一下用于生成跳转指令的 `TranslateBranch()` 函数，先举个简单的例子来说明一下，如图 5.7 所示。对于图 5.7 第 1 至 5 行的代码而言，可以翻译为第 6 至 15 行的中间代码，在第 7 行和第 8 行需要产生 2 条跳转指令，当  $a$  为真时，控制流跳往 `BB_T1`，当  $a$  为假时，第 7 行的条件不成立，不会进行跳转，接着执行第 8 行，跳往 `BB_F1`，这是一个完全正确的翻译方案，不妨称之为“方案 1”。当然，也可以按图 5.24 第 16 至 24 行进行翻译，当  $a$  为假（即  $\neg a$  为真）时跳往 `BB_F2`，但当  $a$  为真时第 17 行的条件不成立，不会进行跳转，接着执行第 19 行的指令，这正是我们想要的结果，即表达式  $a$  为真时执行 `BB_T2` 的代码，而表达式  $a$  为假时执行 `BB_F2` 处的代码，不妨称此方案为“方案 2”。方案 2 比方案 1 不仅减少了代码所占用的空间（减少了一条指令），同时也加快了执行的速度（当  $a$  为假时，没有必要像方案 1 那样需要执行第 7 和第 8 行这两条跳转指令）。

```

7   if(a) goto BB_T1;
8   goto BB_F1;
9   BB_T1:
10  c = 1;
11  goto BB_Next1;
12  BB_F1:
13  c = 2;
14  BB_Next1:
15  ...
16 //////////////方案 2/////////
17  if(!a) goto BB_F2;
18  BB_T2:
19  c = 1;
20  goto BB_Next2;
21  BB_F2:
22  c = 2;
23  BB_Next2:
24  ...

```

图 5.7 生成跳转指令

函数 `TranslateBranch()` 为表达式 `expr` 产生跳转指令时，是按照方案 2 来处理的。我们把表达式分为布尔表达式和算术表达式，对算术表达式而言，先对其求值，其求值后的结果就相当于第 1 行中的 `a`。而对于布尔表达式中的关系运算表达式来说，例如 `if(a>b)` 中的 `a>b`，只要把第 17 行的条件改为 `if(!(a>b)) goto BB_F2` 即可，即

```
if(a <= b) goto BB_F2;
```

换言之，如果函数 `TranslateBranch()` 要处理的是形如 `a+b` 这样的算术表达式，或者 `a>b` 这样的关系运算表达式，由图 5.7 第 17 行可知，我们只需要生成一条跳转指令，给函数 `TranslateBranch()` 传递一个跳转目标作为参数即可，比如第 17 行的 `BB_F2`。比较麻烦的是复杂的短路运算，如图 5.8 所示。

```

1 // hello.c
2 int a,b,c,d;
3 int main(int argc,char * argv[]){
4   if((a && b) || c){
5     d = 1;
6   }else{
7     d = 2;
8   }
9   return 0;
10 }
11 // hello.uil
12 function main
13 if (! a) goto BB2;
14 BB1:
15 if (b) goto BB3;
16 BB2:
17 if (! c) goto BB4;
18 BB3:
19 d = 1;
20 goto BB5;
21 BB4:
22 d = 2;
23 BB5:
24 return 0;
25 ret

```

图 5.8 较复杂的短路运算

当图 5.8 第 4 行的表达式 `((a&&b)||c)` 为真时，需要跳往第 18 行的 `BB3`；为假时需要跳往第 21 行的 `BB4`。而且我们确实在第 15 行用到了跳转目标 `BB3`，又在第 17 行用到了跳转目标 `BB4`。此时，用于产生第 13 至 17 行各跳转指令的函数 `TranslateBranch()` 就需要两个跳转目标作为其参数。因此，其函数接口如下所示，其中 `bt` 和 `bn` 是两个跳转目标。

```
void TranslateBranch(AstExpression expr, BBlock bt, BBlock bn);
```

当参数 `expr` 对应的只是形如 `a+b` 的算术表达式或者形如 `a>b` 的关系运算表达式，我们只需要一个跳转目标 `bt`，即只生成一条跳转指令 “`If(expr) goto bt`”，如下所示。我们期望在条件 `expr` 不成立时会接着执行参数 `bn` 对应的基本块，这就要求 `bn` 对应的基本块紧挨着 “`if(expr) goto bt;`” 这条中间代码，如下所示。

```

if(expr) goto bt
bn:
...
bt:
...

```

不难发现，图 5.7 第 17 至 24 行的“方案 2”完全符合这个模式，其中图 5.7 第 17 行的表达式 `(!a)` 对应上述 `expr`，第 17 行的 `BB_F2` 对应上述 `bt`，而第 18 行的 `BB_T2` 对应上述 `bn`。

当参数 `expr` 为复杂的短路运算表达式时，我们也期望 `bn` 对应的基本块紧挨着“短路运算对应的跳转指令”。不难发现，图 5.8 第 18 行的 **BB3** 对应如下的 `bn`，而第 21 行的 **BB4** 对应如下的 `bt` 基本块，第 13 至 17 行的跳转指令就是“短路运算  $((a \& \& b) \mid\mid c)$  对应的跳转指令”。

翻译方案：

“短路运算  $((a \& \& b) \mid\mid c)$  对应的跳转指令”

`bn:`

...

`bt:`

...

简而言之，在调用函数 `TranslateBranch(expr, bt, bn)` 时，有这么两个约定：

- (1) 当 `expr` 为真时，跳往 `bt` 基本块
- (2) 紧随函数 `TranslateBranch()` 所生成的跳转指令之后的基本块为 `bn`。

有了这样的基础后，就可以来看一下用于生成跳转指令的函数 `TranslateBranch()` 了，如图 5.9 所示，第 46 至 61 行用于处理形如 `a+b` 这样的算术表达式，第 46 行调用 `TranslateExpression()` 函数来对表达式进行求值，其结果存于符号 `src1` 中，当结果为非 0 常数时，则第 1 行的参数 `expr` 的值为永真，我们在第 49 行通过 `GenerateJump()` 生成一条无条件跳转指令，跳往参数 `trueBB` 所对应的基本块。当 `src1` 不为常数时，若其类型为小于 `int` 的整型（例如 `short` 或者 `char`），则先调用第 55 行的 `TranslateCast()` 函数将其提升为 `int` 型，之后再通过第 58 行的 `GenerateBranch()` 函数生成有条件跳转指令，按之前的约定，此处要进行的跳转为“`if(expr) goto trueBB`”，即当 `expr` 不为 0 时，跳往 `trueBB`，第 58 行的 `JNZ` 是 `JumpIfNotZero` 的助记符，第 60 行的注释再次提醒我们，紧接下来就是基本块 `falseBB` 的中间代码。第 21 至 28 行的代码用于翻译关系运算表达式（`>`、`>=`、`<`、`<=`、`==` 和 `!=` 这 6 个关系运算），第 25 行通过调用 `GenerateBranch()` 函数生成有条件跳转指令，跳往 `trueBB` 基本块。第 29 至 42 行的代码用于处理形如“`!!!! kids0`”的表达式，我们先统计！运算符的个数，若为偶数，则可简化为“`kids0`”；若为奇数，则可简化为“`! kids0`”。第 33 行调用 `TranslateExpression()` 函数对表达式 `kids0` 进行求值，其结果存于符号 `src1` 中，之后再根据运算符“`!`”个数的奇偶来选择跳转指令 `JZ` 或者 `JNZ`，第 34 至 40 行的注释，对为何选择 `JZ` 或 `JNZ` 做了解释。需要注意的是，按照我们之前的约定，我们在此处生成的有条件跳转指令要跳往 `trueBB` 基本块。第 41 行的注释又一次的提醒我们，接一下的基本块就是参数 `falseBB` 对应的基本块。

```

1 void TranslateBranch(AstExpression expr,
2                     BBlock trueBB, BBlock falseBB){
3     BBlock rtestBB;
4     Symbol src1, src2;
5     Type ty;
6     switch (expr->op) {
7         case OP_AND: // kids0 && kids1
8             rtestBB = CreateBBlock();
9             TranslateBranch(Not(expr->kids[0]), falseBB, rtestBB);
10            StartBBlock(rtestBB);
11            TranslateBranch(expr->kids[1], trueBB, falseBB);
12            // falseBB:
13            break;
14        case OP_OR: // kids0 || kids1
15            rtestBB = CreateBBlock();
16            TranslateBranch(expr->kids[0], trueBB, rtestBB);
17            StartBBlock(rtestBB);

```

```

18     TranslateBranch(expr->kids[1], trueBB, falseBB);
19     // falseBB:
20     break;
21 case OP_EQUAL: case OP_UNEQUAL: case OP_GREAT:
22 case OP_LESS: case OP_GREAT_EQ: case OP_LESS_EQ:
23     src1 = TranslateExpression(expr->kids[0]);
24     src2 = TranslateExpression(expr->kids[1]);
25     GenerateBranch(expr->kids[0]->ty, trueBB,
26                     OPMMap[expr->op], src1, src2);
27     // falseBB:
28     break;
29 case OP_NOT:
30     { // if(!!!!kids0)
31         int count = 1;
32         // 用于统计!运算符的个数的代码, 略
33         src1 = TranslateExpression(parent->kids[0]);
34         if(count % 2 == 1) { // if(!src1) goto trueBB;
35             // if(src1 is Zero) goto trueBB;
36             GenerateBranch(ty, trueBB, JZ, src1, NULL);
37         } else { // if(src1) goto trueBB
38             // if(src Not Zero) goto trueBB;
39             GenerateBranch(ty, trueBB, JNZ, src1, NULL);
40         }
41     // falseBB:
42     }
43     break;
44 ...
45 default:
46     src1 = TranslateExpression(expr);
47     if (src1->kind == SK_Constant) {
48         if (! (src1->val.i[0] == 0 && src1->val.i[1] == 0)) {
49             GenerateJump(trueBB);
50         }
51     }
52     else{
53         ty = expr->ty;
54         if (ty->categ < INT) {
55             src1 = TranslateCast(T(INT), ty, src1);
56             ty = T(INT);
57         }
58         GenerateBranch(ty, trueBB, JNZ, src1, NULL);
59     }
60     // falseBB:
61     break;
62 }
63 }
```

图 5.9 TranslateBranch()

图 5.9 第 7 至 13 行用于对形如 “kids0 && kids1” 的短路运算进行翻译，其中 kids0 或者 kids1 本身又是表达式，这就需要递归地对其进行处理。按 TranslateBranch() 函数接口的约定，当实际的函数调用为 TranslateBranch(kids0 && kids1, tBB, fBB) 时，所产生的中间代码仍然要满足以下模式。

“当 kids0 && kids1 为真时，跳往 tBB”

---

```
// 上述跳转指令由 TranslateBranch(kids0 && kids1, tBB, fBB) 来产生
fBB:
...
tBB:
...
```

按短路运算的语义要求，当 kids0 为假时，整个表达式(kids0 && kids1)肯定为假，此时我们要生成跳转指令，跳往 fBB，图 5.9 第 9 行用于此目的。如果 kids0 为真，则我们还要接着判断 kids1 是否为真，当 kids1 为真时，整个表达式“kids0 && kids1”就为真，此时需要跳往 tBB，图 5.9 第 11 行用于产生相应的跳转指令；若 kids1 为假，则按照调用 TranslateBranch() 时的约定，接下的基本块就是 fBB，此时控制流会直接流入 fBB 基本块。我们在图 5.9 第 7 至 13 行所产生的中间代码的模式如下所示：

```
“当 kids0 为假时，即 !kids0 为真时，跳往 fBB”
// 上述跳转指令由 TranslateBranch(Not(expr->kids[0]), fBB, fBB) 来产生
rBB:
    “当 kids1 为真时，跳往 tBB”
    // 上述跳转指令由 TranslateBranch(expr->kids[1], tBB, fBB) 来产生
fBB:
    ...
tBB:
...
```

函数 TranslateBranch() 有 3 个参数，在上述代码模式中，我们特意为第 3 个实参添加了下划线，可以发现，调用 TranslateBranch() 函数产生跳转指令后，接下的基本块总是第 3 个实参所对应的基本块，这正是调用 TranslateBranch() 函数时我们要遵守的约定。函数 TranslateBranch() 是一个递归函数，而图 5.9 第 21 至 61 行的代码就充当了递归调用的出口，这些代码通过调用 GenerateBranch() 来产生“有条件跳转指令”，如图 5.9 第 25 行所示，或者通过调用 GenerateJump() 来产生“无条件跳转指令”，如图 5.9 第 49 行所示，但自始自终，我们都遵守另一个约定，即由 GenerateBranch() 或 GenerateJump() 所生成的跳转指令，其跳转目标都是 TranslateBranch() 函数的第 2 个参数 trueBB。

同理，我们不难理解图 5.9 第 14 至 20 行的用于处理“kids0 || kids1”的代码，这里不再啰嗦。至此，我们实现了对布尔表达式的翻译，也初步介绍了用于翻译算术表达式的函数 TranslateExpression()，在下一节中，我们会对其他表达式的翻译进行讨论。

## 5.2.2 公共子表达式

在介绍算术表达式的翻译前，让我们简单重温一下在第 2.5 节介绍过的公共子表达式，相关的数据结构可参考图 2.30 等。在以下代码中，在完成“ $c = a + b;$ ”的计算后，变量 a 和 b 并没有发生变化，因此当面对“ $d = a + b;$ ”时，没有必要重新计算  $a + b$ 。为了突出用于计算  $a + b$  的中间代码的运算符是加法，我们将其记为“ $t1 : a + b$ ”而不使用“ $t1 = a + b$ ”。

```
// C 代码
c = a + b;
d = a + b;
// 中间代码
t1 : a + b;
c = t1;
d = t1;
```

对于 C 语句“ $b = a > 3 ? 50:60;$ ”来说，与其对应的中间代码如下所示，可以发现临时变量  $t0$  会被赋值两次。为了突出此处的运算符确实是赋值号“ $=$ ”，我们把对临时变量  $t0$  的赋值记为“ $t0 = 50$ ”。当然，在对中间代码进行优化时，可把以下对  $t0$  进行赋值的中间代码

改为对 b 进行赋值。为了简单起见，UCC 编译器的优化通常只在一个基本块内进行，而此处的 t0 显然出现在多个基本块中，这就需要我们在生成中间代码时做一些特殊处理，以便优化时可对“t0=50;”指令做修改。产生这样困境的原因在于 UCC 编译器在翻译表达式“a > 3? 50:60”时，并没有考虑其所处的上下文，而总是将条件表达式的结果先存放到一个临时变量中。

```
// b = a > 3? 50:60 对应的中间代码
if (a <= 3) goto BB2;
BB1:
    t0 = 50;      //优化后改为 b = 50;
    goto BB3;
BB2:
    t0 = 60;      //优化后改为 b = 60;
BB3:
    b = t0;      //优化后可删去
```

UCC 编译器在翻译函数调用时，也没有考虑函数调用所处的上下文，总是先把函数调用的返回值存于临时变量中，因此我们就会得到以下两条中间代码。在优化阶段，我们完全可以把这两条中间代码改为“num: f();”。这就需要我们删去“num = t1;”，同时把指令“t1: f();”改为“num:f();”。由于函数调用是有副作用的，因此函数调用 f() 不可以作为公共子表达式使用。若 num 又是一个临时变量，若还存在形如“num = 50;”这样的 MOV 指令，则我们还要考虑对“num:f();”和“num=50;”做进一步的优化。在遇到形如“b = a > 3? f():50;”的 C 语句时，就会出现这样的情况。因此，在为函数调用生成中间代码“t1:f()”时，我们也需要做一些特殊的预处理，以便对中间代码进行修改。

```
t1 : f();
num = t1;
```

在打印出来的中间代码里，对临时变量的赋值操作是用 ‘=’ 还是 ‘:’，其实只是为了方便中间代码的阅读。UCC 编译器中的函数 GenerateAssign() 用于生成形如“t1:a+b;”的中间代码，该中间代码的运算符实际上是 ADD，其中的冒号很清楚地告诉我们这不是一条 MOV 指令，而是 ADD 指令。而函数 GenerateMove() 则用于生成形如“t0 = 50;”的中间代码，运算符是 MOV，其中的 “=” 一目了然地告诉我们这是一条 MOV 指令。函数 GenerateFunctionCall() 用于生成形如“t1 : f();”的中间代码，该中间代码的运算符实际上是 CALL。这几个函数的代码如图 5.10 所示，第 1 至 12 行是函数 GenerateMove() 的代码，第 2 行用于生成一条中间代码，第 3 行把“源操作数 src 和目的操作数 dst”的引用计数加 1，第 4 至 5 行对中间代码进行初始化，通过第 6 行的 AppendInst() 函数把 MOV 指令（形如“t1 = 50;”）添加到当前基本块中。如果目的操作数 dst 是变量（全局、静态或局部变量），由于当前生成的 MOV 指令会对 dst 重新赋值，这就使得以 dst 作为操作数的公共子表达式失效，第 8 行的 TrackValueChange() 就会沿着图 2.30 中的 uses 链表来完成这个工作。如果 dst 是临时变量，为了能在后续优化时，能对当前 MOV 指令进行修改，我们在第 11 行传递给 DefineTemp 函数的第 3 个实参（即 inst）是 MOV 指令的首地址，而不是算术运算的操作数。与之形成对比的是用于产生公共子表达式（形如“t1:a+b”）的函数 GenerateAssign()，如图 5.10 第 28 至 41 行所示，我们在第 40 行传递给 DefineTemp() 函数的实参是“目的操作数 dst、运算符的编码 opcode、源操作数 src1 和源操作数 src2”。图 5.10 第 13 至 27 行的 GenerateFunctionCall() 函数用于生成 CALL 指令（形如“t1:f();”），第 13 行的 recv 对应函数返回值 t1，第 14 行的 faddr 相当于函数的首地址 f，第 14 行的 args 向量用于存放多个实参，第 16 至 20 行会把这些符号对象的引用计数加 1，第 21 至 24 行用于初始化 CALL 指令，并添加到当前基本块中。在后续优化时，可能要修改当前 CALL 指令，因此在第 26 行调用 DefineTemp() 函数时，第 3 个实参仍然是 CALL 指令的首地址 inst，这与 MOV 指令的情况

类似。在第 10 行调用 DefineTemp() 函数是为了把“对同一临时变量进行赋值的多条 MOV 指令”链接到一起，以便后续的优化。而在第 40 行调用 DefineTemp() 函数，则确实为了创建一个 struct valueDef 对象来表示公共子表达式。

```

1 void GenerateMove(Type ty, Symbol dst, Symbol src){
2     IRIInst inst;    ALLOC(inst);
3     dst->ref++;    src->ref++;    inst->ty = ty;
4     inst->opcode = MOV;    inst->opds[0] = dst;
5     inst->opds[1] = src;    inst->opds[2] = NULL;
6     AppendInst(inst);
7     if (dst->kind == SK_Variable){
8         TrackValueChange(dst);
9     }else if (dst->kind == SK_Temp){
10        DefineTemp(dst, MOV, (Symbol)inst, NULL);
11    }
12 }
13 void GenerateFunctionCall(Type ty, Symbol recv,
14                           Symbol faddr, Vector args){
15    ILArg p;    IRIInst inst;    ALLOC(inst);
16    if (recv) recv->ref++;
17    faddr->ref++;
18    FOR_EACH_ITEM(ILArg, p, args)
19        p->sym->ref++;
20    ENDFOR
21    inst->ty = ty;    inst->opcode = CALL;
22    inst->opds[0] = recv;    inst->opds[1] = faddr;
23    inst->opds[2] = (Symbol)args;
24    AppendInst(inst);
25    if (recv != NULL)
26        DefineTemp(recv, CALL, (Symbol)inst, NULL);
27 }
28 void GenerateAssign(Type ty, Symbol dst,
29                      int opcode, Symbol src1, Symbol src2){
30    IRIInst inst;
31    assert(dst->kind == SK_Temp);
32    ALLOC(inst); dst->ref++;    src1->ref++;
33    if (src2) src2->ref++;
34    inst->ty = ty;
35    inst->opcode = opcode;
36    inst->opds[0] = dst;
37    inst->opds[1] = src1;
38    inst->opds[2] = src2;
39    AppendInst(inst);
40    DefineTemp(dst, opcode, src1, src2);
41 }
42 void DefineTemp(Symbol t, int op,
43                  Symbol src1, Symbol src2){
44    ValueDef def;
45    ALLOC(def);
46    def->dst = t;    def->op = op;
47    def->src1 = src1;    def->src2 = src2;
48    def->ownBB = CurrentBB;
49    assert(t->kind == SK_Temp);
50    if (op == MOV || op == CALL) {

```

```

51     def->link = AsVar(t)->def;
52     AsVar(t)->def = def;
53     return;
54 }
55 if (src1->kind == SK_Variable) {
56     TrackValueUse(src1, def);
57 }
58 if (src2 && src2->kind == SK_Variable) {
59     TrackValueUse(src2, def);
60 }
61 AsVar(t)->def = def;
62 }
```

图 5.10 GenerateMove()

图 5.10 第 42 至 62 行的函数 DefineTemp() 可用来创建一个 struct valueDef 对象，该对象描述了一个公共子表达式，第 44 至 48 行对其进行初始化。如果当前指令是 MOV 或 CALL 指令时，通过第 51 至 52 行的链表插入操作，我们把对同一个临时变量进行赋值的若干条指令链接到一起。例如，在介绍“ $b=a > 3? 50:60$ ”的中间代码时，我们遇到了“ $t0=50;$ ”和“ $t0=60;$ ”这两条指令，图 5.10 第 51 至 52 行会把这两条 MOV 指令链到一起，链首存放在  $t0$  对应符号对象的 def 域中。之后在执行优化函数 PeepHole() 时，若遇到中间代码 “ $b = t0;$ ”，就可通过  $t0$  对应符号对象的 def 域，找到这些中间指令，并把其中的  $t0$  都改为  $b$ ，从而就可以得到 “ $b=50;$ ” 和 “ $b=60;$ ” 这两条优化后的指令，之后还可删去 “ $b = t0;$ ”。当源操作数 src1 是“局部、静态或全局变量”，我们要在 src1 变量的 uses 域所指向的链表上添加一个 struct valueUse 对象，用来记录 src1 会在公共子表达式 ( $t: \text{src1} \text{ op } \text{src2}$ ) 中被使用，这个工作由图 5.10 第 56 行的 TrackValueUse() 函数来实现。当然如果存在源操作数 src2，我们也做类似处理，图 5.10 第 58 至 60 行用于此目的。函数 PeepHole() 可对 MOV 指令和 CALL 指令进行优化，如图 5.11 所示。

```

1 static void PeepHole(BBlock bb) {
2     IRIInst inst = bb->insth.next;
3     IRIInst ninst;
4     while (inst != &bb->insth) {
5         ninst = inst->next;
6         if ((inst->opcode == CALL) ||
7             && ninst->opcode == MOV
8             && inst->opds[0] == ninst->opds[1]
9             && inst->opds[0]->ref == 2) {
10        /*****
11        t1:f();      //当前指令 inst
12        num = t1;    //下一条指令 ninst
13        经此处代码优化为:
14        num:f();
15        *****/
16        inst->opds[0]->ref -= 2;
17        inst->opds[0] = ninst->opds[0];
18        inst->next = ninst->next;
19        ninst->next->prev = inst;
20        bb->ninst--;
21        if (ninst->opds[0]->kind == SK_Temp) {
22            DefineTemp(ninst->opds[0],
23                        inst->opcode, (Symbol)inst, NULL);
24        }
25    }
26 }
```

```

24         }
25     }else if (inst->opcode == MOV
26             && inst->opds[1]->kind == SK_Temp) {
27         ****
28         t0 = 50;
29         ....
30         t0 = 60;
31         ...
32         b = t0;      //当前指令 inst
33     经此处优化为:
34         b = 50;
35         ...
36         b = 60;
37         ...
38         b = t0; //删去此条指令
39     ****
40     ValueDef def = AsVar(inst->opds[1])->def;
41     IRInst p;
42     if(def && (def->op == MOV || def->op == CALL)) {
43         while (def != NULL) {
44             p = (IRInst)def->src1;
45             p->opds[0]->ref--;
46             inst->opds[0]->ref++;
47             p->opds[0] = inst->opds[0];
48             def = def->link;
49         }
50         inst->opds[0]->ref--;
51         inst->opds[1]->ref--;
52         // 若 b 也是一个临时变量
53         if(inst->opds[0]->kind == SK_Temp) {
54             //把原先对 t0 和 b 进行赋值的指令都链在一起
55             AppendVarDefList(AsVar(inst->opds[0]),
56                               AsVar(inst->opds[1]));
57         }
58         inst->prev->next = inst->next;
59         inst->next->prev = inst->prev;
60         bb->ninst--;
61     }
62 }
63 }

```

图 5.11 PeepHole()

图 5.11 第 6 至 24 行的代码用于对 CALL 指令进行优化，我们想把“形如第 11 至 12 行注释里的中间代码”优化为第 14 行的中间代码，第 16 行用于把临时变量 t1 的引用计数减 2，第 17 行实现了把“t1:f();”改为“num:f();”，第 18 至 20 行用于删除指令“num = t1;”。当 num 本身又是临时变量时，我们需要把指令“num:f()”添加到 num 对应符号对象的 def 域中，以便后续的进一步优化，这是通过在第 22 行调用 DefineTemp() 函数来实现。图 5.11 第 25 至 61 行用于对 MOV 指令进行优化，我们希望能把“第 28 至 32 行的中间代码”优化为第 34 至 38 行的代码，注释中的例子就是前文的 C 语句“`b = a > 3? 50:60;`”对应的中间代码。当前 MOV 指令形如第 32 行的“`b=t0;`”，其中 t0 为临时变量，由于在这情况下，对 t0 进行赋值的指令（第 28 行的“`t0=50;`”和第 30 行的“`t0=60;`”）都不在当前基本块中，我们可通过第 40 行的代码，取出我们之前在 def 域中保存的 MOV 或 CALL 指令链表，第 40

至 47 行的 while 循环用于把链表中的形如 “t0=50;” 的指令改为 “b=50;”。我们在第 45 行要把 t0 的引用次数减 1，在第 46 行把 b 的引用次数加 1。由于我们要删除第 38 行的“b=t0;”，就需要在第 50 和 51 行把 b 和 t0 的引用次数都减 1，第 58 至 60 行从当前基本块中删除了该指令。如果 b 本身也是个临时变量，我们就把原先所有对 b 进行赋值和对 t0 进行赋值的指令链接到一起，为对 b 的进一步优化做好准备，第 55 行的函数 AppendVarDefList() 实现了这两个链表的合并操作（优化前对 b 进行赋值的指令构成一个链表，而对 t0 进行赋值的指令又构成另外一个链表，合并之后，新的链表中就可能有 CALL 指令，也可能有 MOV 指令，例如 “b = a > 3? f():50;” 对应的中间代码）。函数 AppendVarDefList() 的代码并不复杂，我们从略。

公共子表达式 “t1:a+b” 中的临时变量 t1 则由 C 编译器产生，并不是由 C 程序员给出。而源操作数 a 和 b 可能是临时变量，也可能是由 C 程序员命名的变量。全局变量或静态变量对应“全局静态数据区”中的一块内存单元，而局部变量或者形式参数则对应“栈区”中的一块内存单元。临时变量通常只用于暂存一个计算结果，对应的是“栈区”中的一块内存单元或者直接对应 C 编译器分配的一个寄存器。这些临时变量名对 C 程序员是不可见的，C 程序员不可能对其进行赋值。

由于 C 语言中可以取 a 或 b 的地址，之后再通过地址去访问 a 或 b 对应的内存单元，而不必通过变量名 a 或 b 来访问，因此，要检测操作数 a 或 b 是否被改动过，并不是件容易的事情，这需要进行较复杂的别名分析（Aliase Analysis）。为简单起见，UCC 编译器采取了保守的策略，即一个变量 a 若被进行过“取地址”运算（即&a），则认为 a 可能发生变化，公共子表达式 a+b 不再有效。

在 C 语言中，对数组元素和结构体成员的访问方式也较为灵活，UCC 编译器为了简单起见，也认为这些操作数被进行了“取地址”操作，即不再重用含有这些操作数的表达式。图 5.12 中的例子对此进行了说明，虽然第 20 行只是对 a 进行取地址，并没有通过\*ptr 来改变 a 的值，但 UCC 编译器保守地认为 a 的值会发生变化，因此对第 21 行的 a+b 进行重新计算。而含有结构体成员 dt.a 或者数组元素 arr[0] 的表达式，也不被当作公共子表达式来处理。在第 23 行对 dt.a+dt.b 进行了重新计算，与之对应的中间代码在第 48 行，其中的 dt[0] 对应的就是 dt.a，而 dt[4] 对应的就是 dt.b。在中间代码层次，通过保存在符号表中的类型信息，我们可以知道 dt.a 在结构体对象 dt 中偏移为 0，而 dt.b 在结构体对象 dt 中的偏移为 4。这是一种“基址 base + 偏移 offset”的寻址模式，符号 dt 相当于基址，而 0 或 4 为偏移。在图 5.12 第 25 行，我们也对 C 表达式 arr[0]+arr[1] 重新计算，与之对应的中间代码在第 52 行。在中间代码层次，我们已经把 C 程序员编写的第 25 行的 arr[1]，改用“基址 base + 偏移 offset”的形式来表示，即表示为第 52 行的 arr[4]，其中 arr 相当于基址，而 4 相当于偏移。不过，对于第 9 行的数组 int arr2[3][5] 来说，由于 C 程序员在第 26 行用 arr2[i][2] 这样的方式来访问数组元素，其中 i 为变量，而 2 为常量，这就意味着对数组元素的寻址要按“基址+常量偏移+可变偏移”来进行，其中常量偏移为 2\*sizeof(int)，即 8；可变偏移则为 i\*sizeof(int)\*5，即 i\*20，第 54 至 57 行用于计算出 arr2[i][2] 的地址并存于临时变量 t12 中。对图 5.12 第 58 行的中间代码 “\*t12 = 30;” 而言，若把 t12 的值存于 CPU 的寄存器中，我们就可以通过寄存器间接寻址来对 arr2[i][2] 进行赋值操作。

```

1 // hello.c
2 typedef struct{
3     int a;
4     int b;
5     int num[4];
6 }Data;
7 Data dt;
8 int arr[4];
9 int arr2[3][5];
10 int a,b,c,i;
11 int * ptr;
12 int main(int argc,char * argv[]){
13     // (0: dt)
14     Data dt1 = dt;
15     // (0:1)-->(4:2)-->(8:3)
16     Data dt2 = {1,2,3};

```

```

17 // (0: 1.23)                               40 t0 : a + b;
18 double d = 1.23;                           41 c = t0;
19 c = a+b;                                 42 t1 :&a;
20 ptr = &a;                                43 ptr = t1;
21 c = a+b;                                 44 t2 : a + b;
22 c = dt.a + dt.b;                         45 c = t2;
23 c = dt.a + dt.b;                         46 t4 : dt[0] + dt[4];
24 c = arr[0]+arr[1];                       47 c = t4;
25 c = arr[0]+arr[1];                       48 t5 : dt[0] + dt[4];
26 arr2[i][2] = 30;                          49 c = t5;
27 //dt.num is dt[8]                         50 t7 : arr[0] + arr[4];
28 //dt.num[3] is dt[8+12]                   51 c = t7;
29 dt.num[3] = 50;                           52 t8 : arr[0] + arr[4];
30 return 0;                                53 c = t8;
31 }                                         54 t9 : i * 20;
32 // hello.i                                55 t10 :&arr2;
33 function main() {                         56 t11 : t9 + 8;
34   dt1 = dt;                             57 t12 : t10 + t11;
35   dt2 : 24;                            58 *t12 = 30;
36   dt2[0] = 1;                           59 dt[20] = 50;
37   dt2[4] = 2;                           60 return 0;
38   dt2[8] = 3;                           61 ret
39   d = 1.23;

```

图 5.12 取地址与偏移

我们再来看一下图 5.12 第 29 行的 `dt.num[3]`, `dt.num` 在结构体对象 `dt` 中的偏移为 8, 而 `num[3]` 在数组 `dt.num` 中的偏移为 12, 两者相加, 可得到 `dt.num[3]` 在结构体对象 `dt` 中的偏移为 20。因此, 与第 29 行 `dt.num[3]` 对应的是第 59 行的 `dt[20]`, 我们仍然是用“基址+偏移”的模式来访问 `dt.num[3]`。

对图 5.12 第 16 行的局部变量 `dt2` 的初始化而言, 在第 4.4 节对初始化进行语义检查时, 我们已介绍过如下所示的 `struct initData` 结构体, 第 15 行的注释表示由 3 个 `struct initData` 对象构成的链表, 其中的(8:3)表示我们要用表达式 3, 初始化局部变量 `dt2` 从偏移 8 开始的内存单元, 对应的中间代码为图 5.12 第 38 行的“`dt2[8] = 3;`”。在中间代码层次, 我们还是用“基址+偏移”的模式来访问要被初始化的内存单元。不过对于局部变量 `dt2` 来说, 由于其对应的内存单元在动态分配的栈区中, 在汇编代码中, 我们就没办法使用变量名 `dt2`, 而是使用形如“`movl $3, -40(%ebp)`”这样的汇编指令, 寄存器 `ebp` 在运行时会指向动态分配的栈区。图 5.12 第 35 行的中间代码“`dt2:24`”表示, 要把对象 `dt2` 所占 24 字节栈空间先清 0, 然后再通过第 36 至 38 行的中间代码对相应偏移位置进行初始化操作。

```

struct initData{
    int offset;
    AstExpression expr;
    initData next;
};

```

这里我们初步介绍了对结构体成员或数组元素进行寻址的概念, 在下一节中, 我们对“偏移”做进一步讨论, 分析 UCC 编译器中与此相关的函数, 如 `Offset()` 等。

### 5.2.3 通过“偏移”访问数组元素和结构体成员

在上一节小节, 我们举例介绍了对“数组元素和结构体成员”的访问, 我们采用的是“基地址+偏移”的模式来计算其内存单元的地址。对于数组元素 `arr2[i][2]` 来说, 数组索引值 `i`

为变量，对应的地址要表达为“基地址+常量偏移+可变偏移”；对于结构体成员 dt.b 来说，其地址可表达为“基地址+常量偏移”。下面，我们还是结合一个简单的例子来说明相关概念，如图 5.13 所示，第 1 至 14 行给出了一个简单的 C 程序，第 16 至 30 行是 UCC 编译器生成的中间代码，第 33 至 46 行是 UCC 编译器生成的汇编代码，而第 49 至 58 则是 GCC 编译器生成的汇编代码。由于第 10 行的 arr[i] 含有“可变偏移”，C 编译器需要生成代码来计算这些偏移，再与数组首地址进行相加。在汇编代码中，用于寻址的指令相当灵活，对于“arr[i]=30;”来说，GCC 所生成的代码就与 UCC 不同，如图 5.13 第 50 至 51 行所示。UCC 编译器采用的是形如第 34 至 38 行的指令，第 34 至 35 行用于计算“可变偏移”，即  $i \times 4$ ，通过把  $i$  左移 2 位来实现，第 36 行通过 leal 指令来取数组 arr 的首地址，第 37 行进行把基地址和偏移相加，所得结果存于寄存器 ecx，之后通过第 38 行的寄存器间接寻址就可完成赋值。第 19 行的中间代码“t1:&arr”对应第 36 行的汇编代码“leal arr,%ecx”。我们还发现，第 11 行的 C 代码“arr[2]=50;”对应的汇编代码为第 40 行的“movl \$50,arr+8”。在汇编代码中出现的符号 arr 可看成是一个地址常量，该 movl 指令把常数 50 送到(arr+8)所对应的内存单元中。

```

1 // hello.c
2 int arr[4];
3 int i;
4 struct Data{
5     int a;
6     int num[4];
7 };
8 struct Data dt;
9 int main(int argc,char * argv[]){
10    arr[i] = 30;
11    arr[2] = 50;
12    dt.num[i] = 60;
13    dt.num[2] = 80;
14 }
15 /******UCC 生成的中间代码*****/
16 function main
17 // arr[i] = 30;
18 t0 : i << 2;
19 t1 : &arr;
20 t2 : t1 + t0;
21 *t2 = 30;
22 // arr[2] = 50;
23 arr[8] = 50;
24 // dt.num[i] = 60;
25 t6 :&dt[4];
26 t7 : t6 + t0;
27 *t7 = 60;
28 // dt.num[2] = 80;
29 dt[12] = 80;
30 ret
31
32 /******UCC 生成的汇编*****/
33 // arr[i] = 30;
34 movl i, %eax
35 shll $2, %eax
36 leal arr, %ecx
37 addl %eax, %ecx
38 movl $30, (%ecx)
39 // arr[2] = 50;
40 movl $50, arr+8
41 // dt.num[i] = 60;
42 leal dt+4, %edx
43 addl %eax, %edx
44 movl $60, (%edx)
45 // dt.num[2] = 80;
46 movl $80, dt+12
47
48 /******GCC 生成的汇编*****/
49 // arr[i] = 30;
50 movl i, %eax
51 movl $30, arr(,%eax,4)
52 // arr[2] = 50;
53 movl $50, arr+8
54 // dt.num[i] = 60;
55 movl i, %eax
56 movl $60, dt+4(,%eax,4)
57 // dt.num[2] = 80;
58 movl $80, dt+12

```

图 5.13 对数组元素的寻址

在中间代码层次，一个符号对象 struct symbol（或其“子类”对象，例如 struct variableSymbol）可作为三地址码中的目的操作数或源操作数。为了能生成形如第 36 行的汇编指令，UCC 编译器需要产生一条形如第 19 行的中间代码“t1:&arr;”，其中的临时变量 t1 存放了数组 arr 的首地址。虽然数组中的内容可能会被修改，数组 arr 的地址在数组生命周期内并不会发生变化，所以&arr 可以当作公共子表达式来使用，当我们在第 11 行遇到另一

棵语法树上的 arr 结点时，我们就可重用临时变量 t1 中的值。如果 t1 对应的寄存器为 eax，在汇编层次，我们可为第 11 行的 C 语句 “arr[2]=50;” 生成以下汇编代码。

```
leal arr, %eax;           //取数组 arr 的地址
addl $8, %eax;           //arr[2]在数组 arr 中的偏移为常量 8
movl $50, (%eax);       //通过寄存器间接寻址来进行赋值
```

这些汇编代码可以实现 C 语句 “arr[2]=50;” 所要求的语义，但并不是很高效，我们可以用图 5.13 第 40 行的 “movl \$50,arr+8” 来实现一样的功能。

在知道基地址和偏移的前提下，我们可以通过 UCC 编译器中的函数 Offset()，产生“访问数组元素或结构体成员”的中间代码，如图 5.13 第 18 至 20 行所示；而函数 AddressOf() 则可以生成第 19 行的 “t1:&arr;” 的取地址指令。

函数 Offset() 的代码如图 5.14 所示，当 C 程序员访问数组元素或结构体成员时，第 2 行的参数 addr 是数组或结构体的基地址，参数 voff 是“可变偏移 (variable offset)”。当访问图 5.13 第 12 行的结构体成员 dt.num 时，由于结构体成员 dt.num 在结构体对象 dt 中的偏移是固定的，此时 voff 参数为 NULL。但在访问 dt.num[i] 时，数组元素 dt.num[i] 在数组 dt.num 中的偏移为(i\*4)，这不是常量，此时 voff 不为 NULL，而访问 arr[i] 时 voff 也不为 NULL。第 2 行的另一个参数 coff 代表“常量偏移 (const offset)”，访问图 5.13 第 11 行的 arr[2] 时，coff 的值为 2\*sizeof(int)，即 8。图 5.14 第 3 至 8 行用于产生中间代码，进行基地址、可变偏移和“常量偏移”这三者的加法运算，得到地址后，再由第 7 行的 Deref() 进行“间接寻址操作”，这样我们就可以为 “arr[i]=30;” 生成如图 5.13 第 18 至 21 行的中间代码。当 C 程序员要访问 arr[2] 时，此时图 5.14 第 8 行注释中的 t1 就对应参数 addr，voff 为 NULL，而 coff 的值为 8，为了能产生形如图 5.13 第 40 行的汇编代码 “movl \$50,arr+8”，而不是生成 “leal arr,%eax; addl \$8,%eax; movl \$50,(%eax);” 这 3 条低效的代码，我们在第 11 行调用 CreateOffset() 函数创建了一个新的符号对象，用来在中间代码层次表示 arr[8] 这样的符号。对于图 5.14 第 13 至 14 注释中所示的代码而言，ptr 是指向 int[4] 数组的指针，C 程序员通过(\*ptr)[2] 来访问数组元素时，UCC 编译器会在调用函数 CheckUnaryExpression() 进行语义检查时，构造出一棵形如([[] ptr 0) 8) 的语法树，在翻译这个语法树时，我们会计算出地址为 ptr，而偏移为 8，此时我们把这两者相加，再通过间接寻址才能访问到相应的数组元素，第 15 行调用的 Deref() 完成此功能。如果我们把 C 程序员写的(\*ptr)[2]，错误地表示为中间代码层次的符号 ptr[8]，则最终生成的汇编代码会是 “movl \$50,ptr+8”。假设数组 arr 的首地址是 10000，而全局变量 ptr 的地址为 20000，则变量 ptr 中的内容为 10000，在此 movl 指令中，ptr 是地址常数 20000，该 movl 指令会把常数 50 传送到地址 20008 对应的内存单元。不过，按 C 的语义，(\*ptr)[2] 实际应访问 C 的数组元素 arr[2]，数组元素 arr[2] 的地址为 10008，因此 “movl \$50,ptr+8” 是条错误的指令。所以，在中间代码层次，我们不可用符号 ptr[8] 来表示相应的数组元素 (\*ptr)[2]，而是要生成的形如 “t5 : ptr + 8; \*t5 = 60;” 的代码，这些中间代码是通过图 5.14 第 15 行的函数 Deref() 来产生，Deref 是 Dereference 的缩写，表示“提领操作”，也有译为“解引用”，实际上进行的操作是“间接寻址”。

```
1 static Symbol Offset(Type ty,
2     Symbol addr, Symbol voff, int coff) {
3     if (voff != NULL) { // arr[i];
4         voff = Simplify(T(POINTER),
5             ADD, voff, IntConstant(coff));
6         addr = Simplify(T(POINTER), ADD, addr, voff);
7         return Deref(ty, addr);
8     } // t1:&arr; (t1+8) ---> arr[8]
9     if (addr->kind == SK_Temp
10         && AsVar(addr)->def->op == ADDR) {
11         return CreateOffset(ty,
```

```

12         AsVar(addr)->def->src1, coff,addr->pcoord);
13 } // int (*ptr)[4] = &arr;
14 // (*ptr)[2] --> ptr[0][8] --> (ptr, 8)
15 return Deref(ty, Simplify(T(POINTER),
16                           ADD, addr, IntConstant(coff)));
17 }
18 Symbol CreateOffset(Type ty, Symbol base,
19                      int coff,Coord pcoord) {
20 VariableSymbol p;
21 if(coff == 0 && //{double d = 2.0; Data dt = dt2;}
22     (IsArithType(base->ty) || (ty == base->ty))){
23     return base;    // use symbol dt, not dt[0]
24 }
25 CALLOC(p);
26 if (base->kind == SK_Offset){//dt.num[2]
27     coff += AsVar(base)->offset;
28     base = base->link;
29 }
30 p->pcoord = pcoord;
31 p->addressed = 1;
32 p->kind = SK_Offset;
33 p->ty = ty;
34 p->link = base;
35 p->offset = coff;
36 p->name = FormatName("%s[%d]", base->name, coff);
37 base->ref++;
38 return (Symbol)p;
39 }
40 Symbol Deref(Type ty, Symbol addr){
41 Symbol tmp;
42 if (addr->kind == SK_Temp
43     && AsVar(addr)->def->op == ADDR) {
44     // t1: &arr
45     // *t1 --> arr
46     return AsVar(addr)->def->src1;
47 } // t3: *t2
48 tmp = CreateTemp(ty);
49 GenerateAssign(ty, tmp, DEREF, addr, NULL);
50 return tmp;
51 }
52 Symbol AddressOf(Symbol p){
53 if (p->kind == SK_Temp && AsVar(p)->def->op == DEREF) {
54     // t3: *t2
55     // &t3 --> t2
56     return AsVar(p)->def->src1;
57 }
58 assert(p->kind != SK_Temp);
59 p->addressed = 1;
60 if (p->kind == SK_Variable){
61     TrackValueChange(p);
62 }
63 return TryAddValue(T(POINTER), ADDR, p, NULL);

```

```
64 }
```

图 5.14 Offset()

当要访问结构体成员 `dt.num`, 或者要访问的数组元素不存在“可变偏移”(例如 `arr[2]`)时, 我们可用图 5.14 第 18 至 39 行的 `CreateOffset()` 来为其创建一个符号对象, 第 18 行的 `base` 代表基址, 第 19 行的 `coff` 代表“常量偏移”。如果偏移 `coff` 为 0, 比如当我们初始化图 5.14 第 21 行注释中的局部变量 `d` 和 `dt` 时, 第 22 行的条件就会成立, 此时我们直接返回 `base` 即可。但是如果我们要访问的是 `dt.a` 时, 按图 5.13 第 4 至 7 行的结构体定义, `dt.a` 在对象 `dt` 中的偏移为 0, 但 `dt.a` 和 `dt` 的类型不一样, 因此我们需要为 `dt.a` 创建一个新的符号对象, 而不能使用和 `dt` 一样的符号对象, 此时第 22 行的条件就不成立。第 25 行用于在堆空间中分配一个符号对象, 第 30 行设置该符号在 C 源代码中的坐标, 第 31 行置标志位 `addressee` 为 1, 表示该对象被进行过“取地址操作”(这样, 数组元素和结构体成员充当表达式的操作数时, 该表达式就不再被当作公共子表达式), 第 32 行设置该符号对象的类型为 `SK_Offset`, 第 33 行设置其类型, 第 34 行保存其基址对应的符号对象, 第 35 行存放常量偏移, 第 36 行用于生成形如“`arr[8]`”的符号名。当第 18 行的参数 `base` 本身就对应一个数组元素或者“结构体成员”时, 例如 `dt.num[2]` 中的 `dt.num`, 对于 `dt.num` 来说, 其基址为 `dt`, 按图 5.13 第 4 至 7 行的定义, `dt.num` 在结构体对象 `dt` 中的偏移为 4, 而数组元素 `dt.num[2]` 在数组 `dt.num` 中的偏移为 8, 在中间代码层次, 我们可以把两者相加, 得到 `dt.num[2]` 在结构体对象中的偏移为 12, 图 5.14 第 26 至 29 行的代码用于完成这些操作。

图 5.14 第 40 行的函数 `Deref()` 主要用来生成一条形如“`t3:*t2`”的间接寻址指令, 其中 `t2` 中存放一个地址, `*t2` 表示取“这个地址对应内存单元中的内容”, 并把该内容存于临时变量 `t3` 中, 符号 `t3` 就作为“间接寻址”的结果返回。当然, 如果第 40 行的参数 `addr` 形如第 44 行的 `t1`, 而 `t1` 由中间代码“`t1:&arr`”创建, 则间接寻址操作 `*t1` 可简化为对 `arr` 的访问。

而图 5.14 第 52 行的函数 `AddressOf()` 用于在必要时生成形如“`t:&num`”的取地址指令, 其中的 `num` 是应是左值(具有 C 程序员可见的地址)。如果第 52 行的参数 `p` 是进行“间接寻址”后所得的结果 `t3`, 其中 `t3` 对应的间接寻址指令为“`t3: *t2`”, 则“取地址操作 `&t3`”可简化为 `t2`, 第 52 至 57 行的 if 语句对此进行判断, 此时直接返回 `t2` 即可。当 `num` 被取地址后, UCC 通过调用第 61 行的 `TrackValueChange()` 函数, 使以 `num` 为操作数的公共子表达式失效。UCC 用这样的策略避免了“别名分析”的复杂过程, 当然这会影响生成代码的质量, UCC 编译器在优化上做得还不够。由于 `num` 在其生命周期内的地址是不会变化的, 所以对 `num` 进行取地址后的值就可以作为公共子表达式使用, 第 63 行调用的 `TryAddValue()` 用于此目的。

理解了图 5.14 中的 `Offset()` 等函数后, 由于处理完了“寻址”的问题, 我们再来看 `tranexpr.c` 中的表达式翻译就会轻松许多。在下一小节中, 我们将对 `tranexpr.c` 中用于翻译结构体成员访问的函数 `CheckMemberAccess()`, 及用于翻译数组元素访问的函数 `CheckPostfixExpression()` 等进行讨论。

## 5.2.4 后缀表达式的翻译

在前面的章节中, 我们介绍了用于对数组元素和结构体成员进行访问的函数 `Offset()`, 其接口如下所示, 参数 `addr` 代表了基地址, 参数 `voff` 代表可变偏移, 而参数 `coff` 则代表常量偏移。

```
Symbol Offset(Type ty, Symbol addr, Symbol voff, int coff);
```

函数 `Offset()` 的基本想法是产生以下中间代码, 我们要先对 `addr`、`voff` 和 `coff` 进行相加, 得到目标地址 (`addr+voff+coff`), 然后再进行“提领”操作。

```
t1: coff+voff;
```

```
t2: addr+t1;           //t2 中存放目标地址
t3: *t2;               //提领操作，即间接寻址
```

在遇到如下 arr[2]这样的只包含常量偏移的数组元素时，我们可把上述 addr 看成是“addr:&arr;”，coff 为 8，而 voff 为 NULL，对 C 语言中数组元素 arr[2]的访问，在中间代码层次可用一个符号“arr[8]”来表示即可，不必产生上述中间代码。Offset()函数内会对这种情况进行判断，然后调用 CreateOffset()函数来得到一个名为“arr[8]”的符号对象。

```
int arr[4];
a = arr[2];
```

当我们遇到形如 ptr->b 的表达式时，我们可把 ptr 看成上述 addr 参数，而成员 b 在结构体中的偏移为常量 4，通过 Offset()函数，我们产生“t1: ptr+4; t2: \*t1;”这样的中间代码，其中的符号 t2 即代表了 C 程序员要访问的 ptr->b。

```
typedef{
    int a; //偏移为 0
    int b; //偏移为 4
} Data ;
Data dt; Data * ptr = &dt; ptr->b = 3; dt.b = 5;
```

而对于 dt.b 的处理，与前文对 arr[2]的处理类似，在中间代码层次我们用符号“dt[4]”来表示即可，不必进行间接寻址。当要访问的结构体成员是位域时，则会复杂一些。

```
struct {
    int a; //偏移为 0
    //以下各成员构成的二进制位串为“b4_b3_b2_b1”，
    //其中，低 6 位为 b1，高 9 位为 b4
    int b1:6; //偏移为 4, pos 为 0
    int b2:8; //偏移为 4, pos 为 6
    int b3:9; //偏移为 4, pos 为 14
    int b4:9; //偏移为 4, pos 为 23
} dt2;
int val;
val = dt2.b2; //读取位域成员
```

对上述结构体对象 dt2 来说，当要读取其成员位域 dt2.b2 时，在中间代码层次可用符号“dt2[4]”来表示 dt2.b2 所在的内存单元。由于 UCC 编译器总是把位域成员存于一个 int 型的整数中，符号“dt2[4]”代表的是一个 32 位的 int 型整数对应的内存单元。为了读取 dt2.b2 的值，我们需要进行移位操作，先把整数 dt2[4]左移 18 位（由 9+9 可得到 18），即把高 18 位的 b3 和 b4 给“挤走”，然后再算术右移 24 位（通过 32-8 可得到 24），把低位的 b1 也“挤走”，此时得到的临时变量 t2 就是程序员需要读取的 dt2.b2，如下所示。

```
//左移 18 位，低 18 位补 0，得到二进制位串 b2_b1_0..0,
t1: dt2[4]<<18;
//算术右移 24 位，得到二进制位串 0...0_b2 或者 s...s_b2，其中 s 为 b2 的符号位，
//若 b2 为无符号数则高 24 位补 0；否则高位补上 b2 的符号位 s
t2: t1>>24;
```

UCC 编译器中的函数 ReadBitField()用于产生这些移位操作，由此可对结构体位域成员进行读操作；而对于位域成员的写操作，我们会在翻译赋值表达式时进行讨论。

我们再举个例子来说明一下函数名，对于如下 C 代码，UCC 编译器在遇到“ptr = f;”中的函数名 f 时要生成一条用于取地址的中间代码“t0:&f”，之后再生成“ptr = t0;”的中间代码，这样在 UCC 生成汇编代码时就会选用“leal f,%eax”来获取函数 f 的首地址并存于 eax 寄存器中，而如果错误地生成形如“ptr = f;”的中间代码，则对应的汇编指令为“movl f,%ebx; movl %ebx,ptr;”，这会错误地把函数 f 代码区的前 4 个字节的内容送到 ptr 中。不过，按汇编中 call 指令的语法，我们可生成形如“call f”的汇编指令来调用函数 f，因此

在遇到 C 语句 “f();” 中的函数名 f 时，我们不必为函数名 f 产生取地址指令，直接用返回函数名 f 即可。

```

void f(void) {
}
void * ptr;
int main(int argc,char * argv[]){
    ptr = f;
    f();
    return 0;
}
// UCC 生成的中间代码和汇编代码如下所示
t0 :&f;           // leal f, %eax ;  不可以用 movl f, %eax
ptr = t0;          // movl %eax, ptr
f();              // call f

```

有了这些基础后，我们就可以来分析一下图 5.15，其中第 1 行的函数 TranslateArrayIndex() 用于翻译数组索引，第 7 至 17 行的 do 循环对常量偏移和可变偏移进行累加，分别存于 coff 和 voff 中，第 18 行调用 TranslateExpression() 来得到数组的首地址，第 19 行调用 Offset() 函数来产生访问数组元素的中间代码。第 22 至 29 行的函数 TranslatePrimaryExpression() 用来翻译基本表达式（常量和 ID 等）。对于数组名和函数名，我们在第 27 行调用 AddressOf() 函数来生成取地址的指令，对于其他的标识符，我们在语义检查时已经查过符号表，此处直接在第 28 行返回 expr 结点中保存的符号即可。第 56 至 64 行的函数 ReadBitField() 用于产生左移和右移运算的中间代码，第 58 行计算左移的位数，第 59 行计算右移的位数，第 60 行调用的 Simplify() 会进行一些简单的优化工作，并把左移和右移的表达式当作公共子表达式来重用，我们已在前面的章节分析过 Simplify() 函数。

```

1 static Symbol TranslateArrayIndex(
2             AstExpression expr){
3     AstExpression p;
4     Symbol addr, dst, voff = NULL;
5     int coff = 0;
6     p = expr;
7     do{// arr[i][2]
8         if (p->kids[1]->op == OP_CONST){
9             coff += p->kids[1]->val.i[0];
10        } else if (voff == NULL){
11            voff = TranslateExpression(p->kids[1]);
12        } else{
13            voff = Simplify(voff->ty, ADD, voff,
14                             TranslateExpression(p->kids[1]));
15        }
16        p = p->kids[0];
17    } while (p->op == OP_INDEX);
18    addr = TranslateExpression(p);
19    dst = Offset(expr->ty, addr, voff, coff);
20    return expr->isarray ? AddressOf(dst) : dst;
21 }
22 static Symbol TranslatePrimaryExpression(
23             AstExpression expr){
24     if (expr->op == OP_CONST) // 123
25         return AddConstant(expr->ty, expr->val);
26     if (expr->isarray || expr->isfunc) // arr
27         return AddressOf(expr->val.p); // t:&arr

```

```

28     return expr->val.p;
29 }
30 static Symbol TranslateMemberAccess(
31     AstExpression expr) {
32     AstExpression p;
33     Field fld;
34     Symbol addr, dst;
35     int coff = 0;
36     p = expr;
37     if (p->op == OP_PTR_MEMBER) { // ptr->a
38         fld = p->val.p;
39         coff = fld->offset;
40         addr = TranslateExpression(expr->kids[0]);
41     } else{
42         while (p->op == OP_MEMBER){ // dt.a.b
43             fld = p->val.p;
44             coff += fld->offset;
45             p = p->kids[0];
46         }
47         addr = AddressOf(TranslateExpression(p));
48     }
49     dst = Offset(expr->ty, addr, NULL, coff);
50     fld = dst->val.p = expr->val.p;
51     if (fld->bits != 0 && expr->lvalue == 0){
52         return ReadBitField(fld, dst);
53     }
54     return expr->isarray ? AddressOf(dst) : dst;
55 }
56 static Symbol ReadBitField(Field fld, Symbol p) {
57     int size = 8 * fld->ty->size;
58     int lbits = size - (fld->bits + fld->pos);
59     int rbits = size - (fld->bits);
60     p = Simplify(T(INT), LSH, p, IntConstant(lbits));
61     p = Simplify(GetBitFieldType(fld),
62                 RSH, p, IntConstant(rbits));
63     return p;
64 }

```

图 5.15 数组元素和结构体成员的翻译

图 5.15 第 30 行的 `TranslateMemberAccess()` 用于为结构体成员的访问生成中间代码，第 37 至 40 行用于计算“形如 `ptr->a` 的结构体成员”的基址址和常量偏移，而第 41 至 47 行则为计算出形如 `dt.a.b` 的结构体成员的基址址和常量偏移。在确定基址址和常量偏移后，我们就可以在第 49 行调用 `Offset()` 函数。如果要对结构体位域成员进行读操作，则第 51 行的条件会成立，此时我们在第 52 行调用 `ReadBitField()` 来产生相应的移位指令。

接下来，我们来讨论形如 “`f(a+b,c*d,e);`” 的函数调用的翻译，相关代码如图 5.16 所示，如果我们在第 7 行遇到的 `expr->kids[0]` 是函数名 `f` 对应的语法树结点，则通过第 7 行的 `TranslateExpression()`，我们实际调用的是图 5.15 第 22 行的 `TranslatePrimaryExpression()` 函数，在这情况下我们不需要为函数名 `f` 产生取地址指令，直接返回 `f` 即可。通过在图 5.16 第 6 行设置 `isfunc` 为 0，我们可以使图 5.15 第 26 行的 `if` 条件不成立。第 9 至 15 行的 `while` 循环用来依次对各个实参表达式进行计算，并把计算所得的结果通过第 13 行插入到向量 `args` 中，如果返回值不是 `void` 类型，我们就在第 18 行创建一个临时变量用来存放函数的返回值，第

20 行调用 GenerateFunctionCall() 函数来生成 CALL 指令，我们已在前面的章节中分析过这个函数，这里不再重复。

```

1 static Symbol TranslateFunctionCall(AstExpression expr) {
2     AstExpression arg;
3     Symbol faddr, recv;
4     ILArg ilarg;
5     Vector args = CreateVector(4);
6     expr->kids[0]->isfunc = 0;
7     faddr = TranslateExpression(expr->kids[0]);
8     arg = expr->kids[1];
9     while (arg) {
10         ALLOC(ilarg);
11         ilarg->sym = TranslateExpression(arg);
12         ilarg->ty = arg->ty;
13         INSERT_ITEM(args, ilarg);
14         arg = (AstExpression)arg->next;
15     }
16     recv = NULL;
17     if (expr->ty->categ != VOID) {
18         recv = CreateTemp(expr->ty);
19     }
20     GenerateFunctionCall(expr->ty, recv, faddr, args);
21     return recv;
22 }
```

图 5.16 TranslateFunctionCall()

另一种后缀表达式是形如“ $a++$ 和 $a--$ ”的表达式，UCC 编译器在语义检查时已把它们分别转换为 $a+=1$ 和 $a-=1$ 来处理，而 $+=$ 和 $-=$ 是赋值运算符，我们会在讨论赋值表达式的翻译时进行介绍。

## 5.2.5 赋值表达式的翻译

在这一小节中，我们来讨论一下赋值表达式的翻译，例如“ $a=a-b;$ ”或者“ $a += b;$ ”等。在图 5.17 的下方我们给出了表达式“ $a+=b;$ ”在语法分析后和语义检查后的语法树，右下侧的语法树相当于是表达式 $a = a'+b$ ，但表达式中的 $a$  和 $a'$ 对应同一个语法树结点。按 C 的语义，语句“ $a+=b;$ ”中的 $a$ 只能被求值一次，而不能求值两次。例如，若 $a$ 结点对应一个函数调用 $(*f())$ ，则语句“ $(*f()) += b;$ ”中的函数 $f$ 只被调用一次。在图 5.17 的上方是 C 语句“ $a = a+b;$ ”的语法树，在该树中有两个语法树结点 $a$ ，若把 $a$ 改为表达式 $(*f())$ ，则我们需要为语句“ $(*f()) = (*f()) + b;$ ”产生两次对函数 $f$ 的调用。

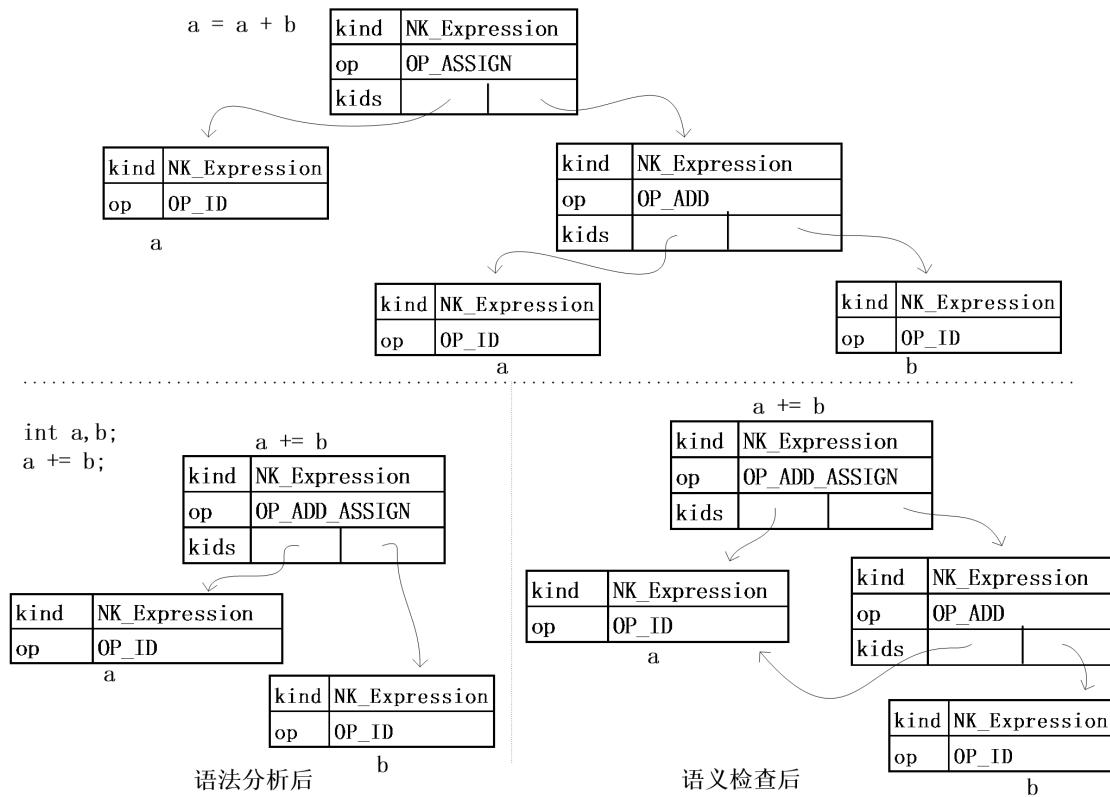


图 5.17 赋值表达式的语法树

对于赋值语句“`a = b;`”来说，当 `a` 是结构体位域成员时，对 `a` 的写操作会较复杂。下面还是举一个例子来说明。以下结构体对象 `dt` 中的位域 `b1`、`b2`、`b3` 和 `b4` 共占用 32 位内存（即 4 字节），其所处偏移为 4，在中间代码层次，我们可用符号“`dt[4]`”来表示该内存单元。

```
struct Data{
    int a; //偏移为 0
    //以下各成员构成的二进制位串为“b4_b3_b2_b1”，
    //其中，低 8 位为 b1，高 4 位为 b4
    int b1:8; //偏移为 4, pos 为 0
    int b2:16; //偏移为 4, pos 为 8
    int b3:4; //偏移为 4, pos 为 24
    int b4:4; //偏移为 4, pos 为 28
    int d; //偏移为 8
} dt;
struct Data * ptr = &dt;
ptr->d = 30;
dt.b2 = val;
```

当 C 程序员要通过“`dt.b2 = val;`”来对位域 `b2` 进行赋值时，我们可按以下步骤来实现：

(1) 按以下中间代码来构造二进制位串“`b4_b3_val_b1`”。

```
t1: val << 8;           // LSH 指令，左移 8 位
t2: t1 & 0x00FFFF00;   // BAND 指令，得到二进制位串“0...0_0...0_val_0...0”
t3: dt[4] & 0xFF0000FF; // BAND 指令，得到二进制位串“b4_b3_0...0_b1”
t4: t2 | t3;           // BOR 指令，得到二进制位串“b4_b3_val_b1”
```

(2) 把二进制位串“`b4_b3_val_b1`”赋值给 `dt[4]`，即可实现对 `b2` 的赋值，同时保持 `b1`、`b3` 和 `b4` 不会发生变化。

```
dt[4] = t4;
```

当我们面对“`ptr->d = 30;`”时，UCC 编译器为 `ptr->d` 产生的中间代码如下所示，由于 `t7` 是个临时变量，如果对 `t7` 进行赋值，则 `ptr->num` 的值并没有发生变化。我们要对 `t6` 所指向的内存单元进行赋值，才能改变 `ptr->num` 的值。

```
t5: ptr
t6: t5 + 8;      //成员 d 的偏移为 0
t7: *t6;
```

此时我们可产生形如“(IMOV, t6, 30);”的中间代码，其中 IMOV 表示把 30 赋值给 `t6` 所指向的内存单元，而不是把 30 赋值给 `t6`。UCC 编译器的 `GenerateIndirectMove()` 函数用于产生 IMOV 指令。若 `t6` 的值存放在寄存器 `eax` 中，则最终生成的汇编代码可以是“`movl $30, (%eax)`”，对应的中间代码可表示为：

```
*t6 = 30;          //而不是 t6 = 30;
```

有了这些基础之后，我们就可以来分析一下赋值表达式的翻译，如图 5.18 所示。第 29 至 60 行的 `WriteBitFiled()` 函数用于实现对位域成员 “`dt.b2`” 的赋值操作，按前文的介绍，对于形如 “`dt.b2 = val;`” 表达式来说，我们要构造二进制位串 “`b4_b3_val_b1`” 来对符号 “`dt[4]`” 进行赋值。图 5.18 第 46 至 50 行分别用于产生前文的 “左移 LSH”、“按位与 BAND”、“按位与 BAND” 和 “按位或 BOR” 这 4 条指令，从而得到二进制位串 “`b4_b3_val_b1`”。当 C 程序员编写的是形如 “`ptr->b2=val;`” 的语句，我们需要在第 54 行产生一条 IMOV 指令（形如 “`*t = val;`”）来实现赋值，这需要进行间接寻址；而当 C 程序员编写的是形如 “`dt.b2 = val;`” 的语句时，我们在第 57 行产生 MOV 指令（形如 “`dt[4] = val;`”）即可，不必进行间接寻址。对赋值表达式 “`dt.b2 = val`” 来说，虽然按 C 的语义，整个表达式 “`dt.b2 = val`” 不可充当左值来使用，例如 “`(dt.b2 = val) = 3;`” 是非法的，但在赋值操作后，`dt.b2` 的值即为整个表达式 “`dt.b2 = val`” 的值。因此，我们在第 59 行调用函数 `ReadBitField()` 来读取 `dt.b2` 的值，用来作为整个表达式 “`dt.b2 = val`” 的值。当 “`dt.b2 = val;`” 中的 `val` 为常数时，我们可在编译时进行一些常量折叠的处理，避免生成一些不必要的指令，函数 `WriteBitFiled()` 中的其他代码主要用于这些情况，结合图 5.18 中的注释不难理解相关代码，我们就不再啰嗦。

```
1 static Symbol TranslateAssignmentExpression(
2     AstExpression expr){
3     Symbol dst, src;
4     Field fld = NULL;
5     dst = TranslateExpression(expr->kids[0]);
6     fld = dst->val.p;
7     if (expr->op != OP_ASSIGN){ // k0 += k1;
8         expr->kids[0]->op = OP_ID;//避免对 k0 多次求值
9         expr->kids[0]->val.p =
10             expr->kids[0]->bitfld ?
11                 ReadBitField(fld, dst) : dst;
12     }
13     src = TranslateExpression(expr->kids[1]);
14     if (expr->kids[0]->bitfld){//对位域成员赋值
15         return WriteBitField(fld, dst, src);
16     } else if (dst->kind == SK_Temp
17         && AsVar(dst)->def->op == DEREF){//ptr->d = 30;
18         // dst: *addr;
19         // dst = src; ---> (IMOV, addr, src)
20         Symbol addr = AsVar(dst)->def->src1;
21         GenerateIndirectMove(expr->ty, addr, src);
22         dst = Deref(expr->ty, addr);
23     }else{ // k0 = k1;
24         // (MOV, dst, src);
```

```

25     GenerateMove(expr->ty, dst, src);
26 }
27 return dst;
28 }
29 static Symbol WriteBitField(Field fld,
30     Symbol dst, Symbol src){// dt.b2 = val;
31 int fmask = (1 << fld->bits) - 1;//0x0000FFFF
32 int mask = fmask << fld->pos;// 0x00FFFF00
33 Symbol p;
34 .....
35 if (src->kind == SK_Constant
36     && src->val.i[0] == 0){ //0xFF0000FF
37     p = Simplify(T(INT),
38         BAND, dst, IntConstant(~mask));
39 } else if (src->kind == SK_Constant
40     && (src->val.i[0] & fmask) == fmask){//0x0000FFFF
41     p = Simplify(T(INT), BOR, dst, IntConstant(mask));
42 }else{
43     if (src->kind == SK_Constant){
44         src = IntConstant((src->val.i[0] << fld->pos) & mask);
45     } else{
46         src = Simplify(T(INT), LSH, src, IntConstant(fld->pos));
47         src = Simplify(T(INT), BAND, src, IntConstant(mask));
48     }
49     p = Simplify(T(INT), BAND, dst, IntConstant(~mask));
50     p = Simplify(T(INT), BOR, p, src);
51 }
52 if (dst->kind == SK_Temp && AsVar(dst)->def->op == DEREF) {
53     Symbol addr = AsVar(dst)->def->src1;
54     GenerateIndirectMove(fld->ty, addr, p);
55     dst = Deref(fld->ty, addr);
56 } else{
57     GenerateMove(fld->ty, dst, p);
58 }
59 return ReadBitField(fld, dst);
60 }

```

图 5.18 TranslateAssignmentExpression()

当我们面对的是形如“ $a+=b;$ ”的赋值表达式时，UCC 编译器在图 5.18 第 5 行调用函数 `TranslateAssignmentExpression()`，递归地对与  $a$  对应的左子树进行翻译。由图 5.17 右下方的语法树可知，为了避免对  $a$  对应的子树进行重复翻译，我们在图 5.18 第 8 行把左子树  $a$  的根结点当作一个标识符结点，这样即使当  $a$  为( $*f()$ )时，我们也只会产生一次函数调用  $f$ 。另一个需要特殊处理的是当  $a$  为“ $dt.b2$ ”这样的位域成员时，若我们面对的是形如“ $dt.b2 += b;$ ”这样的赋值表达式，则图 5.18 第 5 行真正调用的函数是 `TranslateMemberAccess()`。当位域成员  $dt.b2$  处于赋值号的左侧时，在 `TranslateMemberAccess()` 函数中，我们并未调用 `ReadBitFiled()` 进行读操作，因此在图 5.18 第 10 行我们还要对此进行判断。对于“ $dt.b2 += b;$ ”而言，当我们第一次访问到  $dt.b2$  的子树时，在图 5.18 第 5 行的函数调用返回后， $dst$  对应的符号为“ $dt[4]$ ”。由于  $dt.b2$  是位域，我们需要在第 11 行调用 `ReadBitFiled()` 函数进行读操作，这将产生以下中间代码，其中符号“ $t2$ ”就保存了  $dt.b2$  的值，我们需要在第 9 行把符号“ $t2$ ”保存在  $dt.b2$  对应子树的根结点，当我们第二次访问到  $dt.b2$  对应的子树时，其根结点已被当成一个标识符结点，其符号名为“ $t2$ ”，这样我们就可在稍后产生形如“ $t3: t2+b$ ”的加法

指令。

```
t1 : dt[4] << 8;
t2 : t1 >> 16;
```

图 5.18 第 13 行递归地调用 TranslateExpression() 来翻译赋值运算符的右子树，第 15 行调用 WriteBitFiled() 来实现对位域成员的写操作，第 16 至 22 行用于处理形如 “ptr->d=30;” 的赋值运算，我们需要在第 21 行产生形如 “\*t = 30;” 的 IMOV 指令用于间接寻址，而第 25 行产生形如 “a=b;” 的 MOV 指令。

在此基础上，我们再来理解 “a++” 或 “++a” 这样的运算就轻松了许多，图 5.19 给出了这两个表达式在语义检查后的语法树。若 a 在执行自加运算前的值为 val，按照 C 语言的语义，经过 a++ 和 ++a 的运算后，a 的值都变为(val+1)，但表达式 “a++” 的值为 val，而表达式 “++a” 的值为(val+1)。需要注意的是，在 C 语言中，“a++” 和 “++a” 都不可以作为左值来使用，即 “(a++) = 3;” 和 “(++a) = 5;” 都是非法的；但在 C++ 语言中，“++a” 可充当左值使用，即 “(++a) = 5;” 是合法的，但 “(a++) = 3;” 是非法的。

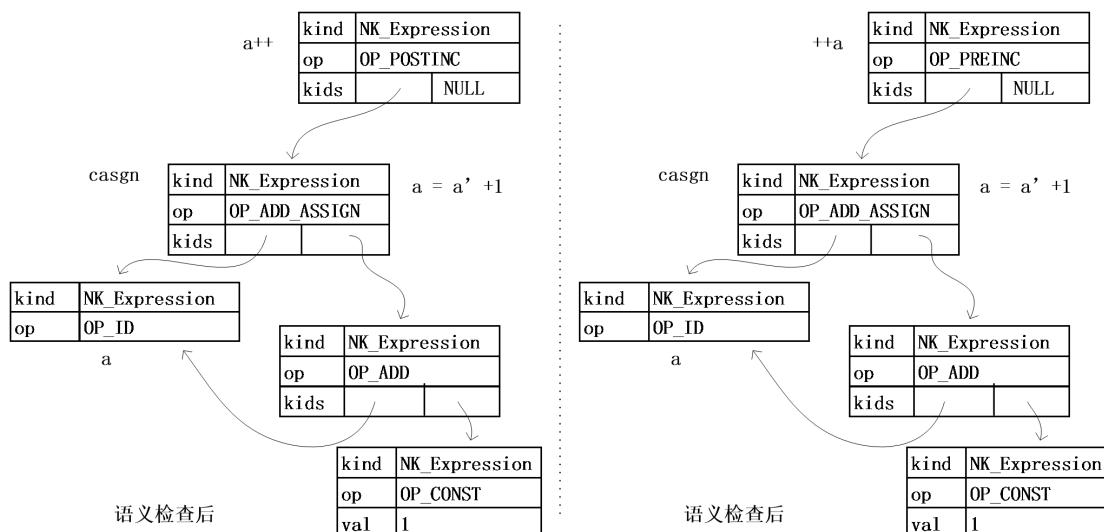


图 5.19 a++ 和 ++a 的语法树

结合图 5.19，我们来讨论对自加运算进行翻译的函数 TranslateIncrement()，如图 5.20 所示。对于 “++a” 来说，我们在图 5.20 第 22 行调用 TranslateExpression(casgn) 即可完成翻译。但对于 “a++” 来说，我们要先计算出图 5.19 结点 a 的值，该值即为整个表达式 “a++” 的值，之后再进行  $a = a' + 1$  的运算来使 a 的值加 1。图 5.20 第 6 行调用 TranslateExpression() 函数完成了对 a 的翻译，第 7 行把该子树置为标识符结点，也是为了避免对 a 的多次求值。

```

1 static Symbol TranslateIncrement(AstExpression expr) {
2     AstExpression casgn;
3     Symbol p;
4     Field fld = NULL;
5     casgn = expr->kids[0];
6     p = TranslateExpression(casgn->kids[0]);
7     casgn->kids[0]->op = OP_ID;
8     casgn->kids[0]->val.p = p;
9     fld = p->val.p;
10    if (expr->op == OP_POSTINC || expr->op == OP_POSTDEC) { // a++, a--
11        Symbol ret;
12        ret = p;
13        if (casgn->kids[0]->bitfld) { // a 是位域成员
14            ret = ReadBitField(fld, p);
15        } else if (p->kind != SK_Temp) { // a 不是临时变量

```

```

16         ret = CreateTemp(expr->ty);
17         GenerateMove(expr->ty, ret, p);
18     }
19     TranslateExpression(casgn);
20     return ret;
21 }
22 return TranslateExpression(casgn);
23 }

```

图 5.20 TranslateIncrement()

对于形如“`a++`”的表达式来说，`a`对应的语法树结点有这样几种情况：

(1) 结点 `a` 是结构体位域成员。此时第 13 行的 if 条件会成立，我们需要通过第 14 行的 `ReadBitField()` 函数来读取该位域的值并存放到 `ret` 中，`ret` 就是整个表达式“`a++`”的值。

(2) 结点 `a` 不是位域成员，也不是临时变量。此时第 15 行的条件会成立，我们在第 16 行创建一个临时变量 `ret` 来保存整个表达式“`a++`”的值。

(3) 结点 `a` 不是位域成员，是临时变量。此时第 13 至 18 行的 if 条件都不成立，我们处理的是形如“(ptr->d)++”的表达式。在 UCC 编译器内部，在翻译形如“`ptr->d`”的表达式时，需要进行提领操作，会产生形如“`t1:*t2`”的临时变量 `t1`。我们在第 12 行就已经把符号 `t1` 赋值给 `ret`，符号 `t1` 代表的是还没有进行自加运算前的 `ptr->d` 的值。此时，在第 19 行真正调用的函数是 `TranslateAssignmentExpression()`，用于实现对结点 `a` 的加 1 运算，第 20 行返回“`a++`”整个表达式的值 `ret`。

在下一节中，我们将分析一下 `tranexpr.c` 中的其他函数。

## 5.2.6 一元表达式及其他表达式的翻译

在这一小节中，我们先来讨论一下一元表达式的翻译，我们先举个例子来说明一下。在以下 C 程序中，符号 `arr` 被声明为 `int[3][5]` 的数组类型，UCC 编译器在语义检查后，为表达式“`**arr=30;`”构造的语法树为“(=([]([] arr 0) 0) 30)”，在中间代码生成阶段，对应的中间代码为“`arr[0] = 30;`”。由这个例子可见，并不是所有出现在 C 程序中的指针运算符“`*`”都会对应“提领操作”。

```

int arr[3][5];
int * ptr = &arr[0][0];
void f(void) {
    ** arr = 30;
    *ptr = 50;
}

```

对于 C 表达式“`*ptr=50;`”，UCC 编译器在语义检查后构造的语法树为“(= (\* ptr) 50)”，在中间代码生成阶段，UCC 编译器需要为该语法树中的“`*`”运算符产生“提领指令”。一元表达式的翻译如图 5.21 所示，第 4 行调用 `TranslateBranchExpression()` 函数来计算形如“`!a`”的布尔表达式的算术值（0 或者 1），第 7 行调用函数 `TranslateIncrement()` 来翻译表达式“`++a`”或者“`--a`”，第 14 行调用的 `AddressOf()` 用于产生“取地址指令”，第 16 行的 `Deref()` 函数用于产生“提领指令”。我们已在前面的章节中分析过这几个函数，这里不再重复。

```

1 static Symbol TranslateUnaryExpression(AstExpression expr) {
2     Symbol src;
3     if (expr->op == OP_NOT) {// !a
4         return TranslateBranchExpression(expr);
5     }
6     if (expr->op == OP_PREINC || expr->op == OP_PREDEC) {// ++a, --a
7         return TranslateIncrement(expr);
}

```

```

8     }
9     src = TranslateExpression(expr->kids[0]);
10    switch (expr->op) {
11      case OP_CAST:           // (int) a
12        return TranslateCast(expr->ty, expr->kids[0]->ty, src);
13      case OP_ADDRESS:        // &a
14        return AddressOf(src);
15      case OP_DEREF:          // *a
16        return Deref(expr->ty, src);
17      case OP_NEG:            // -a ~a
18        return Simplify(expr->ty, OPMMap[expr->op], src, NULL);
19      default:
20        assert(0);
21        return NULL;
22    }
23  }
24 }
```

图 5.21 TranslateUnaryExpression()

图 5.21 第 12 行调用 TranslateCast() 来为形如 “(int) a” 的表达式产生转型指令，该函数并不复杂，我们也从略。第 17 至 19 行用于翻译形如 “-a” 和 “~a”的表达式，我们在第 19 行调用 Simplify() 函数做一些编译时的代数简化，该函数也已在前面的章节中做过分析。

接下来我们再分析条件表达式的翻译，以下给出了一段 C 程序及其对应的中间代码，我们生成临时一个临时变量 t0 用来存放 b+2 或 b+3 的值，然后再把 t0 赋值给变量 c。

```

int a ,b,c;
c = a > 0? b+2: b+3;
////////////////////////////对应的中间代码////////////////////////////
if (a <= 0) goto BB7;
BB6:           // trueBB
t1 : b + 2;
t0 = t1;
goto BB8;
BB7:           // falseBB
t2 : b + 3;
t0 = t2;
BB8:           // nextBB
c = t0;
```

UCC 编译器中的函数 TranslateConditionalExpression() 用于产生上述中间代码，如图 5.22 所示，第 6 行调用的 CreateTemp() 函数用于产生上述临时变量 t0，第 8 行的 trueBB 对应“BB6”，而第 9 行的 falseBB 对应上述“BB7”，第 10 行的 nextBB 对应“BB8”。第 11 行调用 TranslateBranch() 函数来产生上述跳转指令 “if(a <= 0) goto BB7;”，第 13 行递归地调用 TranslateExpression() 函数用来翻译 “b+2”，而第 15 行的 GenerateMove() 用来产生 MOV 指令 “t0 = t1;”，第 17 行产生无条件跳转指令 “goto BB8;”。第 18 至 22 行的代码用于翻译 “b+3” 并产生相应的标号 “BB7” 和 “BB8”。

```

1 static Symbol TranslateConditionalExpression(AstExpression expr) {
2   Symbol t, t1, t2;
3   BBlock trueBB, falseBB, nextBB;
4   t = NULL;
5   if (expr->ty->categ != VOID) {
6     t = CreateTemp(expr->ty);
7   }
8   trueBB = CreateBBlock();
```

```

9   falseBB = CreateBBlock();
10  nextBB = CreateBBlock();
11  TranslateBranch(Not(expr->kids[0]), falseBB, trueBB);
12  StartBBlock(trueBB);
13  t1 = TranslateExpression(expr->kids[1]->kids[0]);
14  if (t1 != NULL) {
15      GenerateMove(expr->ty, t, t1);
16  }
17  GenerateJump(nextBB);
18  StartBBlock(falseBB);
19  t2 = TranslateExpression(expr->kids[1]->kids[1]);
20  if (t2 != NULL) {
21      GenerateMove(expr->ty, t, t2);
22  }
23  StartBBlock(nextBB);
24  return t;
25 }
```

图 5.22 TranslateConditionalExpression()

我们已在图 5.6 中介绍过二元表达式的翻译函数 TranslateBinaryExpression(), 而逗号表达式的翻译, 可由以下函数 TranslateCommaExpression() 来实现。例如, 对逗号表达式 “a,b,c” 而言, 按照 C 语言的语义, 最后一个表达式 “c” 的值即为整个逗号表达式的值, UCC 编译器通过语义检查会为表达式 “a,b,c” 产生语法树 “(, (, a b) c)”。我们只要递归地调用 TranslateExpression() 函数来翻译其左子树和右子树即可, 如下所示。

```

// 翻译逗号表达式, 我们面对的是形如“(, (, a b) c)”的语法树
static Symbol TranslateCommaExpression(AstExpression expr) {
    // 翻译左子树(, a b)
    TranslateExpression(expr->kids[0]);
    // 翻译右子树 c
    return TranslateExpression(expr->kids[1]);
}
```

至此, 我们完成了对表达式的翻译, 在下一节中, 我们要讨论语句的翻译。

## 5.3 语句的翻译

### 5.3.1 If 语句和复合语句的翻译

我们先简单回顾一下对布尔表达式的翻译, 我们通过调用 TranslateBranch() 函数来产生跳转指令, 从而实现布尔表达式的语义。在使用函数 TranslateBranch(expr, bt, bn) 时, 有这么两个约定:

- (1) 当 expr 为真时, 跳往 bt 基本块;
- (2) 紧随 “函数 TranslateBranch() 所生成的跳转指令” 之后的基本块为 bn。

在表达式的基础上, 我们来讨论一下语句的翻译。图 5.23 用于 if 语句的翻译, 第 4 至 8 行说明了如何翻译 “if(expr) stmt”, 我们需要创建 trueBB 和 nextBB 这两个基本块; 而第 12 至 18 行阐述了如何翻译 “if(expr) stmt else stmt” 语句, 我们需要创建 trueBB、falseBB 和 nextBB 这 3 个基本块。图 5.23 第 29 至 34 行的代码用于翻译 “if(expr) stmt”, 而第 35 至 43 行的代码用于翻译 “if(expr) stmt else stmt”。

```

1 /**
2 * This function translates an if statement.
```

```

3  *
4  * if (expr) stmt is translated into:
5  *   if ! expr goto nextBB
6  * trueBB:
7  *   stmt
8  * nextBB:
9  *   ...
10 /////////////////
11 * if (expr) stmt1 else stmt2 is translated into:
12 *   if ! expr goto falseBB
13 * trueBB:
14 *   stmt1
15 *   goto nextBB
16 * falseBB:
17 *   stmt2
18 * nextBB:
19 *   ...
20 */
21 static void TranslateIfStatement(
22         AstStatement stmt){
23     AstIfStatement ifStmt = AsIf(stmt);
24     BBlock nextBB;
25     BBlock trueBB;
26     BBlock falseBB;
27     nextBB = CreateBBlock();
28     trueBB = CreateBBlock();
29     if (ifStmt->elseStmt == NULL) {
30         TranslateBranch(
31             Not(ifStmt->expr), nextBB, trueBB);
32         StartBBlock(trueBB);
33         TranslateStatement(ifStmt->thenStmt);
34     } else {
35         falseBB = CreateBBlock();
36         TranslateBranch(
37             Not(ifStmt->expr), falseBB, trueBB);
38         StartBBlock(trueBB);
39         TranslateStatement(ifStmt->thenStmt);
40         GenerateJump(nextBB);
41         StartBBlock(falseBB);
42         TranslateStatement(ifStmt->elseStmt);
43     }
44     StartBBlock(nextBB);
45 }

```

图 5.23 TranslateIfStatement()

由图 5.23 可以发现，在理解了表达式翻译的前提下，if 语句的翻译就较容易理解了。而其他的控制流语句，包括 do 语句、while 语句和 for 语句也与此类似，我们就不在啰嗦。接下来，我们来讨论复合语句的翻译。语法上，复合语句的产生式如下所示，由可选的声明列表和语句列表构成，其中的声明可以带有初值。

compound-statement → { declaration-list<sub>opt</sub> statement-list<sub>opt</sub>}

由于局部变量的存储空间是运行时在栈中动态分配的，UCC 编译器需要在编译时产生中间代码来实现对局部变量的初始化，而在对应的汇编代码中，由于无法预知相应的存储单元的具体地址，我们只能采用“ebp 寄存器+常量偏移”的模式来对其寻址。程序运行时，

我们会在栈中为被调用的函数分配一块内存，用于存放其形参、局部变量、函数返回地址等信息，这块内存通常被称为“活动记录（activation record）”或者“帧（frame）”。在 x86 平台上，我们会使用 x86 的 `ebp` 寄存器来指向当前函数的活动记录；而“常量偏移”则可由 C 编译器在编译时，根据局部变量声明在函数中出现的先后顺序和所占内存大小来确定。对于以下局部数组 `arr` 的初始化，我们可以先把 `arr` 所占的栈空间清 0，然后再根据“初值及其偏移”来产生初始化相应数组元素的指令。

```
int arr[8] = {10, 20, 30};
```

我们在语义检查时介绍过以下结构体 `struct initData`，其中的 `offset` 即可用于描述偏移，而 `expr` 为初值，`next` 用于构造链表。

```
struct initData{
    int offset;           //偏移
    AstExpression expr;   //初值对应的表达式
    InitData next;
};
```

```
typedef struct initData * InitData;
```

上述初值{10,20,30}经语义检查后，可得到以下链表，在中间代码生成阶段，我们可由该链表产生“对相应数组元素进行初始化”的中间代码。

```
(表达式 10, 偏移 0) --> (表达式 20, 偏移 4) --> (表达式 30, 偏移 8)
/////////对应的中间代码//////////
```

```
arr : 40;          //把 arr 所占 40 字节栈内存清 0
arr[0] = 10;
arr[4] = 20;
arr[8] = 30;
```

而到了汇编层次，局部变量的名字不再可用，我们改用“`ebp` 寄存器+常量偏移”的方式来访问局部变量。对应的汇编代码如下所示：

```
pushl $40
pushl $0
leal -40(%ebp), %eax
pushl %eax
call memset           //对数组 arr 清 0，相当于调用 memset(&arr[0], 0, 40);
addl $12, %esp
movl $10, -40(%ebp) //arr[0] = 10;
movl $20, -36(%ebp)
movl $30, -32(%ebp)
```

对于以下局部数组 `num` 来说，由于初值{1,2,3,4}覆盖了数组 `num` 所占的所有内存空间，我们并不需要额外产生对数组 `num` 进行清 0 的指令。

```
int num[4] = {1,2,3,4};
```

有了这些基础后，我们就来看一下复合语句的翻译，如图 5.24 所示，第 1 至 9 行的函数 `HasPaddingSpace()` 用来判断初值之间是否有空隙，例如上述数组 `arr` 的初值之间就有空隙，而数组 `num` 没有。这可通过累加各初值所占内存大小，然后再与整个局部对象所占内存大小作比较，即可得到。

```
1 static int HasPaddingSpace(VariableSymbol v) {
2     int initSize = 0;
3     InitData initd = v->idata;
4     while(initd){
5         initSize += initd->expr->ty->size;
6         initd = initd->next;
7     }
8     return initSize != v->ty->size;
```

```

9  }
10 static void TranslateCompoundStatement(
11     AstStatement stmt){
12     AstCompoundStatement compStmt = AsComp(stmt);
13     AstNode p = compStmt->stmts;
14     Vector ilocals = compStmt->ilocals;
15     Symbol v;
16     int i;
17     for (i = 0; i < LEN(ilocals); ++i){
18         InitData initd;
19         Type ty;
20         Symbol dst, src;
21         int size;
22         v = GET_ITEM(ilocals, i);
23         initd = AsVar(v)->idata;
24         size = 0;
25         if(HasPaddingSpace(AsVar(v))){
26             GenerateClear(v, v->ty->size);
27         }
28         while(initd){
29             ty = initd->expr->ty;
30             if (initd->expr->op == OP_STR){
31                 String str = initd->expr->val.p;
32                 src = AddString(
33                     ArrayOf(str->len + 1, ty->bty),
34                     str,&initd->expr->coord);
35             }else{
36                 src = TranslateExpression(initd->expr);
37             }
38             dst = CreateOffset(
39                 ty, v, initd->offset,v->pcoord);
40             GenerateMove(ty, dst, src);
41             initd = initd->next;
42         }
43     }
44     while (p){
45         TranslateStatement((AstStatement)p);
46         p = p->next;
47     }
48 }

```

图 5.24 TranslateCompoundStatement()

图 5.24 第 17 至 43 行的 for 循环用来处理各个局部变量，第 26 行调用 `GenerateClear()` 函数，产生“对整个局部对象清 0”的指令。第 30 行判断初值表达式的类型，当表达式为字符串时，我们在第 32 行调用 `AddString()` 得到一个代表该字符串的符号；对于其他表达式，则在第 36 行调用 `TranslateExpression()` 来翻译。第 38 行调用 `CreateOffset()` 函数生成一个形如前文“`arr[4]`”的符号，第 40 行通过调用 `GenerateMove()` 函数产生形如“`arr[4] = 20;`”的中间代码。图 5.24 第 44 至 47 行的 while 循环依次调用 `TranslateStatement()` 来实现对语句列表的翻译。

在下一小节中，我们来讨论一下 `switch` 语句的翻译，`transtmt.c` 中的不少代码与此相关，由于使用了跳转表的技术，UCC 编译器为 `switch` 语句产生的代码还是相当高效的。

### 5.3.2. Switch 语句的翻译

在这一小节中，我们来讨论一下 switch 语句的翻译，switch 语句的产生式如下所示。

switch-statement:

```
switch ( expression ) statement
```

当 C 程序员编写出如下代码时，UCC 编译器会在语义检查阶段进行报错 “error:The break shall appear in a switch or loop”，从语法上来看，以下 “case 3: b = 30;” 是 case 语句，而 “break;” 并不是 case 语句的一部分。按 switch 语句的产生式，以下 “switch(a) case 3: b = 30;” 构成一条完整的 switch 语句，而 “break;” 则不在 switch 语句内。

switch(a)

```
case 3: b = 30; break;
```

虽然按 switch 语句的产生式，其中的 statement 部分并不一定是复合语句，但通常 C 程序员在编写代码时，会将此部分写为一个复合语句，例如：

```
switch(a) {
    case 3: b = 30; break;
}
```

此时，复合语句中的 “break;” 就包含在 switch 语句中，这就可以符合 C 语言的语义要求，即 “break 语句要被包含在 switch 语句或者循环语句” 中。到了中间代码生成阶段，我们面对的已经是 “没有语法或语义错误”的语法树。我们还是举一个例子来说明 switch 语句的翻译，如图 5.25 所示，第 4 至 17 行是一个 switch 语句，与之对应的中间代码在第 23 至 40 行。我们注意到，第 28 行是一条 “间接跳转” 指令，我们根据(a-1)的值来索引跳转表 “[BB4,BB8,BB6,BB12,BB10,]”，从而得到相应的跳转目标，例如当 a 为 1 时，我们得到的跳转目标为 BB4。当 a 的值为 4 时，由于 4 并不落在 {1, 3, 2, 5} 中，对应的跳转目标 BB12 用于跳出 switch 语句。通过跳转表的技术来翻译 switch 语句，可以使得生成的汇编代码在运行时不需要进行过多的比较，从而加快代码的执行速度，不过，跳转表本身需要占用内存空间，这是一种 “以空间换时间”的方法。第 44 至 53 行是对应的部分汇编代码，第 44 至 49 行的跳转表 switchTable1 在数据区中，其中存放的是跳转目标的首地址（代码区中的地址），而第 50 至 53 行则根据(a-1)的值来查表，再进行跳转。

1 //////////////hello.c//////////	21 //////////////////中间代码/////////
2 int a,b;	22 function main
3 int main(int argc,char * argv[]){	23 if (a < 1) goto BB12;
4 switch(a){	24 BB1:
5 case 1:	25 if (a > 5) goto BB12;
6     b = 10;	26 BB2:
7     break;	27 t0 : a - 1;
8 case 3:	28 goto (BB4,BB8,BB6,BB12,BB10,) [t0];
9     b = 30;	29 BB4:
10    break;	30 b = 10;
11 case 2:	31 goto BB12;
12     b = 20;	32 BB6:
13     break;	33 b = 30;
14 case 5:	34 goto BB12;
15     b = 50;	35 BB8:
16     break;	36 b = 20;
17 }	37 goto BB12;
18 b = 60;	38 BB10:
19 return 0;	39 b = 50;
20 }	40 goto BB12;

```

41 BB12:          48      .long   .BB12
42 b = 60;        49      .long   .BB10
43 ////////////////汇编代码/////////////
44 .data           50 .text
45 swtchTable1:.long .BB4       51 movl a, %eax
46             .long .BB8       52 subl $1, %eax
47             .long .BB6       53 jmp *swtchTable1(%eax,4)

```

图 5.25 switch 语句的例子

在图 5.25 的例子中，各 case 语句中的常量为{1, 3, 2, 5}，这些整数在数值上相差不大，如果我们遇到的是{1, 2, 20000, 50}，相应的跳转表就得有 20000 个表项，其中大部分的表项存放的是形如“BB12”这样的用于跳出 switch 语句的地址，这需要耗费相当大的内存，此时我们不再构造一张跳转表，而是想把{1, 2, 20000, 50}分成几段，每一段内的各个数值彼此接近。例如，我们可把{1, 2, 20000, 50}分成 3 段{{1,2}, {50}, {20000}}。为了衡量各数值彼此接近的程度，UCC 编译器引入了“Case 语句密度”的概念：

$$\text{Case 语句密度} = \text{Case 语句数量} / \text{区间大小}$$

例如，对于{1,2,20000,50}来说，其密度为(4/(20000-1))，这有点像人口密度的概念，反映了人口的密集程度，而{1,2}的密度为 2/(2-1)。为了方便代码生成，UCC 编译器在语法分析时，会把在同一 switch 语句的各 case 语句，按出现的先后顺序进行排列。而为了统计“Case 语句密度”，在语义检查时，会按各 case 语句的常量表达式数值，从小到大进行排列。因此，每条 case 语句都会出现在两个链表中，UCC 编译器用结构体 struct astCaseStatement 来描述一条 case 语句，其中的 next 和 nextCase 域用于构造这样的两个链表。

```

struct astCaseStatement{
    //按 case 语句在源代码中出现的先后顺序排列
    struct astNode *next;
    //按 case 语句的表达式数值从小到大排列
    struct astCaseStatement *nextCase;
    ...
}

```

当面对从小到大排列的{1, 2, 50, 20000}时，UCC 编译器要求每段的密度要大于 1/2，根据这个经验值，我们可以按以下步骤进行分段，

- (1) 第 1 个 case 语句就是一段，即{1}。
- (2) 把第 2 个 case 语句加入{1}，则有{1,2}，其密度为 2。
- (3) 若把第 3 个 case 语句加入{1,2}，则有{1, 2, 50}，其密度为 3/49，该值小于 1/2，因此我们新创建一段来存放 50，即有了{{1, 2}, {50}}。
- (4) 若把第 4 个 case 语句加入{50}，则有{50,20000}，其密度也小于 1/2，因此我们再新创建一段来存放 20000，即有了{{1, 2}, {50}, {20000}}。

若有一个整数 val，我们要判断 val 是落在哪一段中，为了减少比较的次数，我们可以采用二分查找的方法，即先看看 val 是否落在中间的那一段{50}，如果比中间段更小，就再看看是否能在左侧的段{1, 2}中；若更大，则看看是否落在右侧的段{20000}中。按这样思路，我们可以产生如图 5.26 第 23 至 38 行的比较和跳转操作，而第 39 至 50 行的中间代码仍然保持 C 源程序中 case 语句的先后顺序。我们还注意到，当某段中的 case 语句多于 1 条时，UCC 编译器会通过跳转表来进行跳转，如图 5.26 第 34 行所示。如果某段中只有一条 case 语句且其左侧或右侧的段还没有被比较过，例如第 23 行的{50}，我们可产生形如第 23 至 27 行的代码，即要处理“小于”、“大于”和“落在段中”这 3 种情况；而如果该段的左侧和右侧的其他段都已被处理过，例如第 36 行的{20000}，我们可产生形如第 36 至 38 行的代码即可，此时只要处理“等于”和“不等于”这 2 种情况。

```

1 int a,b;          2 int main(int argc,char * argv[]) {

```

```

3   switch(a) {
4     case 1:
5       b = 10;
6       break;
7     case 2:
8       b = 20;
9       break;
10    case 20000:
11      b = 20000;
12      break;
13    case 50:
14      b = 50;
15      break;
16  }
17  b = 60;
18  printf("main()\n");
19  return 0;
20 }

21 ////////////////中间代码/////
22
23  if (a < 50) goto BB4;//{50}
24  BB1:
25  if (a > 50) goto BB8;
26  BB2:
27  goto BB17; //跳往 case 50
28  BB4:
29  if (a < 1) goto BB19;//{1,2}
30  BB5:
31  if (a > 2) goto BB19;
32  BB6:
33  t0 : a - 1;
34  goto (BB11,BB13,) [t0];//case 1/2
35  BB8:
36  if (a != 20000) goto BB19;//{20000}
37  BB9:
38  goto BB15; //跳往 case 20000
39  BB11:           //case 1
40  b = 10;
41  goto BB19;
42  BB13:           // case 2
43  b = 20;
44  goto BB19;
45  BB15:           // case 20000
46  b = 20000;
47  goto BB19;
48  BB17:           // case 50
49  b = 50;
50  goto BB19;
51  BB19:           //switch之后的语句
52  b = 60;

```

图 5.26 Case 语句密度与二分查找

我们再举一个例子来说明分段中可能出现的情况，例如当 UCC 编译器面对从小到大排列的 case 语句{0, 1, 4, 9, 10, 11}时，我们可以按以下步骤进行分段：

- (1) 加入 case 0，得到{0}。
- (2) 加入 case 1，得到{0, 1}，密度为 2。
- (3) 加入 case 4，得到{0,1,4}，密度为 3/4。
- (4) case 9 单独构成一段，得到{{0,1,4},{9}}。
- (5) 加入 case 10，得到{{0,1,4},{9,10}}。
- (6) 加入 case 11，先得到{{0,1,4},{9,10,11}}，由于  $6/11$  仍大于  $1/2$ ，因此我们把这两段合并为{0,1,4,9,10,11}。

为了描述“段”的概念，UCC 编译器引入了 switchBucket 结构体，Bucket 是“桶”的意思，我们用“Bucket”来存放处于同一段中的各条 case 语句。每个段对应一个桶，多个桶构成了一条链表。而每个桶内又可包含多条 case 语句，这些 case 语句又可构成一条链表，switchBucket 结构体如下所示：

```

typedef struct switchBucket{ //用于描述形如{0, 1, 4}这样的段
    int ncase;           //桶内 case 语句的条数，例如 3
    int minVal;          //桶内 case 语句表达式的最小值，例如 0
    int maxVal;          //桶内 case 语句表达式的最大值，例如 4
    AstCaseStatement cases; //桶内 case 语句链表的链首
    AstCaseStatement *tail; //指向链尾，便于插入操作
    struct switchBucket *prev; //用于组成由各个“桶”对象构成的链
} *SwitchBucket;

```

为了方便对各 SwitchBucket 进行二分查找，UCC 编译器除了把各 switchBucket 对象通过上述 prev 域构成一条链表外，还会用一个数组来存放各 switchBucket 的首地址。例如，对于{{1, 2}, {50}, {20000}}来说，我们可用以下伪代码来表示相关结构：

```

SwitchBucket ptr1 = {1, 2};
SwitchBucket ptr2 = {50};
SwitchBucket ptr3 = {20000}
其链表结构为:
{1, 2} ---> {50} ---> {20000}
其数组结构为:
SwitchBucket bucketArray[] = {ptr1, ptr2, ptr3};

```

有了这样的基础后，我们就可以来看一下 switch 语句的翻译，如图 5.27 所示，第 7 行用于翻译 switch 语句中的表达式，第 11 至 31 行把各 case 语句加入到相应的桶中。由于每个 case 语句都是控制流的跳转目标，因此每个 case 语句都对应一个基本块，第 14 行调用 CreateBlock() 函数创建了这些基本块。第 16 行的 if 条件用于判断“当前桶中的 case 语句密度是否大于 1/2”，第 21 行通过调用 MergeSwitchBucket() 函数进行相邻桶的合并，由此可把前文的 {0, 1, 4} 和 {9, 10, 11} 这两个桶合并为 {0, 1, 4, 9, 10, 11}。当 Case 语句密度小于或等于 1/2 时，我们通过第 23 至 28 行创建一个新桶。第 32 至 39 行用于创建桶指针数组，便于进行二分查找。当 Switch 语句的表达式的值不与任何 case 匹配时，控制流要么进入 default 语句（在 C 程序员提供 default 语句时），要么跳出 switch 语句（在 C 程序员没有编写 default 语句时），第 40 至 46 行会对此进行处理。图 5.27 第 48 行调用 TranslateSwitchBuckets() 函数，用于产生形如“图 5.26 第 23 至 38 行的用于比较和跳转”的中间代码。如果 switch 语句确实包含 case 或者 default 语句，第 54 行就递归地调用 TranslateStatement() 函数，翻译候选式“switch(expr) statement”中的 statement。

```

1 static void TranslateSwitchStatement(AstStatement stmt) {
2     AstSwitchStatement swtchStmt = AsSwitch(stmt);
3     AstCaseStatement p, q;    SwitchBucket bucket, b;
4     SwitchBucket *bucketArray; int i, val;
5     Symbol sym;
6     //翻译 switch(expr) statement 中的表达式
7     sym = TranslateExpression(swtchStmt->expr);
8     ....
9     bucket = b = NULL;
10    p = swtchStmt->cases;
11    //创建由 SwitchBucket 桶构成的链表结构
12    while (p){
13        q = p;      p = p->nextCase;
14        q->respBB = CreateBBBlock();
15        val = q->expr->val.i[0];
16        if (bucket &&
17            (bucket->nCase + 1) * 2 > (val - bucket->minVal)){
18            bucket->nCase++;
19            bucket->maxVal = val;           *bucket->tail = q;
20            bucket->tail = &(q->nextCase);
21            swtchStmt->nBucket -= MergeSwitchBucket(&bucket);
22        }else{
23            ALLOC(b);
24            b->cases = q;          b->nCase = 1;
25            b->minVal = b->maxVal = val;
26            b->tail = &(q->nextCase);
27            b->prev = bucket;       bucket = b;
28            swtchStmt->nBucket++;
29        }
30    }
31    swtchStmt->buckets = bucket;

```

```

32 //创建数组结构, 用于二分查找
33 bucketArray = HeapAllocate(CurrentHeap,
34     swtchStmt->nbucket * sizeof(SwitchBucket));
35 for (i = swtchStmt->nbucket - 1; i >= 0; i--) {
36     bucketArray[i] = bucket;
37     *bucket->tail = NULL;
38     bucket = bucket->prev;
39 }
40 swtchStmt->defBB = CreateBBlock();
41 if (swtchStmt->defStmt) {
42     swtchStmt->defStmt->respBB = swtchStmt->defBB;
43     swtchStmt->nextBB = CreateBBlock();
44 } else{
45     swtchStmt->nextBB = swtchStmt->defBB;
46 }
47 //产生比较指令
48 TranslateSwitchBuckets(bucketArray, 0,
49     swtchStmt->nbucket - 1,
50     sym, NULL, swtchStmt->defBB);
51 //翻译 switch(expr) statement 中的语句
52 if(swtchStmt->cases != NULL
53     || swtchStmt->defStmt != NULL) {
54     TranslateStatement(swtchStmt->stmt);
55 }
56 StartBBlock(swtchStmt->nextBB);
57 }

```

图 5.27 TranslateSwitchStatement()

接下来, 我们来分析一下用于产生比较和跳转语句的 TranslateSwitchBuckets() 函数, 其函数接口如下所示:

```

static void TranslateSwitchBuckets(
    //对 bucketArray[left] 至 bucketArray[right] 这几个桶进行处理
    SwitchBucket *bucketArray, int left, int right,
    //符号 choice 代表了 switch(expr) statement 中表达式的值
    Symbol choice,
    //指向当前要处理的 SwichBucket 桶或者为 NULL
    BBlock currBB,
    //参数 defBB 要么是 default 语句对应的基本块(存在 default 语句时)
    //要么是 switch 语句之后的基本块 nextBB(当 default 语句不存在时)
    BBlock defBB
);

```

例如, 在图 5.27 第 48 行我们按以下方式来调用 TranslateSwitchBuckets() 函数, 其中的 swtchStmt->nbucket 代表 switch 语句中 case 的个数, 此处待翻译的各桶的下标从 0 至 (swtchStmt->nbucket - 1)。

```
TranslateSwitchBuckets(bucketArray, 0, swtchStmt->nbucket - 1,
    sym, NULL, swtchStmt->defBB);
```

仍以图 5.26 为例, 对于 {{1, 2}, {50}, {20000}} 而言, 按二分查找的算法, 我们最先处理的是位于中间的桶 {50}, 我们会为之产生以下用于比较和跳转的代码:

```

if (a < 50) goto BB4;          //再去与桶{1,2}比较
BB1:
    if (a > 50) goto BB8;      //再去与桶{20000}比较
BB2:
    goto BB17;                //跳往 case 50

```

图 5.28 给出了 TranslateSwitchBuckets() 函数的代码，第 15 至 18 行用于构造长度为 len 的跳转表，第 19 至 26 行用于对跳转表进行初始化，从而得到形如“(BB11, BB13,)” 的表格，当桶中只有一个 case 语句，且该桶左侧或右侧的其他桶都已被处理完，例如当我们处理桶 {20000} 时，我们可通过第 31 至 32 行产生一条比较指令“if(a != 20000) goto BB19;”，而当我们面对{50}或{1,2}时，则需要通过第 34 至 39 行产生形如“if(a < 50) goto BB4; BB1: if (a > 50) goto BB8;”这样的中间代码。

```

1 static void TranslateSwitchBuckets(
2     SwitchBucket *bucketArray,
3     int left, int right,
4     Symbol choice, BBlock currBB, BBlock defBB) {
5     int mid, len, i;    AstCaseStatement p;
6     BBlock lhalfBB, rhalfBB;
7     BBlock *dstBBS; Symbol index;
8     if (left > right) {
9         return;
10    }
11    mid = (left + right) / 2;
12    lhalfBB = (left > mid - 1) ? defBB : CreateBBlock();
13    rhalfBB = (mid + 1 > right) ? defBB : CreateBBlock();
14    //构造长度为 len 的跳转表
15    len = bucketArray[mid]->maxVal
16                - bucketArray[mid]->minVal + 1;
17    dstBBS = HeapAllocate(
18        CurrentHeap, (len + 1)* sizeof(BBlock));
19    for (i = 0; i < len; ++i)
20        dstBBS[i] = defBB;
21    dstBBS[len] = NULL;
22    p = bucketArray[mid]->cases;
23    while (p) {
24        i = p->expr->val.i[0] - bucketArray[mid]->minVal;
25        dstBBS[i] = p->respBB; p = p->nextCase;
26    }
27    if (currBB != NULL){
28        StartBBlock(currBB);
29    }
30    if(len == 1 && lhalfBB == rhalfBB){
31        GenerateBranch(choice->ty, lhalfBB, JNE,
32                        choice, IntConstant(bucketArray[mid]->minVal));
33    }else{
34        GenerateBranch(choice->ty, lhalfBB, JL,
35                        choice, IntConstant(bucketArray[mid]->minVal));
36        StartBBlock(CreateBBlock());
37        GenerateBranch(choice->ty, rhalfBB,
38                        JG, choice, IntConstant(bucketArray[mid]->maxVal));
39    }
40    StartBBlock(CreateBBlock());
41    if (len != 1){
42        //当桶内 case 个数大于 1 时，通过跳转表进行跳转
43        index = CreateTemp(choice->ty);
44        GenerateAssign(choice->ty, index, SUB,
45                        choice, IntConstant(bucketArray[mid]->minVal));
46        GenerateIndirectJump(dstBBS, len, index);

```

```

47 } else{
48     //直接跳入相应的 case 语句
49     GenerateJump(dstBBS[0]);
50 }
51 StartBBBlock(CreateBBBlock());
52 TranslateSwitchBuckets(bucketArray, left,
53                         mid - 1, choice, lhalfBB, defBB);
54 TranslateSwitchBuckets(bucketArray, mid + 1,
55                         right, choice, rhalfBB, defBB);
56 }

```

图 5.28 TranslateSwitchBuckets()

图 5.28 第 40 至 50 行用于产生跳入相应 case 语句的代码，例如上述用于跳入 case 50 的指令“BB2: goto BB17;”，当跳转表的大小为 1 时，我们只有一个跳转目标，通过第 49 行产生一条无条件跳转指令即可；否则在第 43 至 46 行，通过跳转表来进行跳转，例如我们在图 5.26 中为桶{1,2}产生以下跳转代码：

```

BB6:
t0 : a - 1;
goto (BB11,BB13,) [t0]; //跳往 case 1 或 case 2

```

图 5.28 第 52 至 53 递归地调用 TranslateSwitchBuckets()，对位于当前桶的左侧的各个桶进行处理，而第 54 至 55 行则递归地对右侧各桶进行翻译。

Switch 语句的翻译是所有控制流语句中最复杂的。为了讨论的完整性，我们再给出 UCC 编译器中 for 语句、while 语句和 do 语句的翻译方案，如图 5.29 第 2 至 30 行所示，其他编译器的翻译方案可能与此有所不同，但都要实现相同的语义。图 5.29 第 32 至 41 行用于翻译 break 语句，break 可跳出循环语句或 switch 语句，第 36 和 38 行通过产生无条件跳转指令跳出这些语句。第 42 至 47 行用于翻译 continue 语句，这可通过在第 45 行产生跳转语句，跳入循环语句的 contBB 基本块来实现，例如图 5.29 第 6 行、第 15 行和第 23 行所示的 contBB。第 48 至 56 行用于处理 return 语句，如果有返回值，我们可在第 52 行调用 TranslateExpression() 函数来翻译相应的表达式，并把结果存于 RET 指令中。在 UCC 编译器中，中间代码层次的 RET 指令并不改变控制流，UCC 编译器为简化处理，使每个待翻译的函数都只有唯一的入口基本块 entryBB，及唯一的出口基本块 exitBB，第 54 行会产生无条件跳转指令跳入 exitBB 基本块，即相当于要从函数返回。

```

1 ****
2 do stmt while (expr)
3 Do 语句的翻译方案:
4   loopBB:
5     stmt
6   contBB:
7     if (expr) goto loopBB
8   nextBB: ...
9  /////////////
10 while (expr) stmt
11 While 语句的翻译方案:
12   goto contBB
13   loopBB:
14     stmt
15   contBB:
16     if (expr) goto loopBB
17   nextBB: ...
18 /////////////
19 for (expr1; expr2; expr3) stmt

```

```

20 for 语句的翻译方案:
21     expr1
22     goto testBB
23 loopBB:
24     stmt
25 contBB:
26     expr3
27 testBB:
28     当 expr2 不存在时, 直接 goto loopBB
29     否则, 产生 if expr2 goto loopBB
30 nextBB:    ...
31 ****
32 static void TranslateBreakStatement(
33     AstStatement stmt){
34     AstBreakStatement brkStmt = AsBreak(stmt);
35     if (brkStmt->target->kind == NK_SwitchStatement){
36         GenerateJump(AsSwitch(brkStmt->target)->nextBB);
37     }else{
38         GenerateJump(AsLoop(brkStmt->target)->nextBB);
39     }
40     StartBBlock(CreateBBlock());
41 }
42 static void TranslateContinueStatement(
43     AstStatement stmt){
44     AstContinueStatement contStmt = AsCont(stmt);
45     GenerateJump(contStmt->target->contBB);
46     StartBBlock(CreateBBlock());
47 }
48 static void TranslateReturnStatement(AstStatement stmt){
49     AstReturnStatement retStmt = AsRet(stmt);
50     if (retStmt->expr){
51         GenerateReturn(retStmt->expr->ty,
52                         TranslateExpression(retStmt->expr));
53     }
54     GenerateJump(FSYM->exitBB);
55     StartBBlock(CreateBBlock());
56 }

```

图 5.29 循环语句的翻译方案

至此, 我们完成了对语句的翻译, 为了得到运行时更高效的代码, 还需要对已生成的中间代码进行优化, 例如在图 5.26 第 49 至 52 行, 可以看到以下代码:

```

b = 50;
goto BB19;
BB19:
b = 60;

```

经过优化, 我们可删除其中无用的跳转指令 “`goto BB19;`”, 从而得到:

```

b = 50;
BB19:
b = 60;

```

优化是编译相关领域研发的热点, 与优化相关的理论和技术不仅可用于编译器, 还经常被用于信息安全等领域。UCC 编译器只进行了一些简单的代码优化, 我们会在下一节中进行讨论。

## 5.4 UCC 编译器的优化

### 5.4.1 删 除无用的临时变量和优化跳转目标

UCC 编译器在优化方面做的工作不多，其中与优化有关的函数主要有以下几个：

(1) Symbol Simplify(Type ty, int opcode, Symbol src1, Symbol src2);

用于进行“代数恒等式”的简化，例如表达式“ $a \ll 0$ ”可简化为  $a$ 。

(2) Symbol TryAddValue(Type ty, int op, Symbol src1, Symbol src2);

用于处理“公共子表达式”。

(3) void Optimize(FunctionSymbol fsym);

删除无用的临时变量、优化跳转目标和合并基本块等。

在前面的章节中，我们已经对函数 Simplify() 和 TryAddValue() 做过讨论，在这一小节中，我们主要讨论函数 Optimize()。在中间代码生成阶段，产生了许多临时变量，用来存放表达式的值，在优化时，如果发现有些临时变量的值并没有在其他地方被使用（即该临时变量的引用计数为 1），则可删除该临时变量。如图 5.30 所示，对于第 14 至 18 行的 C 程序而言，优化前的中间代码在第 24 至 28 行，其中的临时变量 t0 至 t4 并没有在其他地方被使用。我们可删除 t1、t2、t3 和 t4，第 31 至 32 行是优化后的中间代码。

```

1 // hello.c
2 typedef struct{
3     int arr[4];
4 }Data;
5 Data GetData(void){
6     Data dt;
7     return dt;
8 }
9 int f(void){
10    return 2015;
11 }
12 int a,b;
13 int main(int argc,char * argv[]){
14     GetData();
15     a|b;// BOR
16     ~a; // BCOM
17     &a; // ADDR
18     f();
19     return 0;
20 }

21 ////////////中间代码///////
22 //优化前的中间代码
23 function main
24     t0 : GetData();
25     t1 : a | b;
26     t2 :~a;
27     t3 :&a;
28     t4 : f();
29 //优化后的中间代码
30 function main
31     t0 : GetData();
32     f();
33 ////////////汇编代码/////
34 //按 GetData(&t0)去调用
35 leal -16(%ebp), %eax
36 pushl %eax
37 call GetData
38 addl $4, %esp
39 //f();
40 call f

```

图 5.30 删 除无用的临时变量

我们注意到，图 5.30 第 24 行的 t0 在优化后仍然得到了保留，其原因在于函数 GetData() 的返回值为结构体对象。为第 24 行的中间代码“t0: GetData();”生成汇编指令时，UCC 编译器将其视为“GetData(&t0);”，我们会在主调函数 main() 的栈中为 t0 分配内存，然后把 t0 的首地址传递给被调函数 GetData()。对 UCC 编译器而言，第 5 行的函数 GetData() 的接口相当于是：

```
void GetData(Data * ptr);
```

按照 C 的语义，函数的返回值可以是整数、浮点数和结构体对象，但不可以是数组。

如果返回值是整数，这些值可以暂存于 X86 的通用寄存器 eax 和 edx 中；而如果返回值是浮点数，则可暂存于浮点协处理器 X87 的寄存器中；但对于图 5.30 第 4 行的结构体来说，其

结构体对象要占 16 字节的空间，如果 CPU 中的寄存器无法存放结构体对象，UCC 编译器在生成汇编代码时，就会隐式地按照 GetData(&t0) 来处理第 31 行的函数调用。在 x86 平台上的 C 调用约定，要规定“函数调用的返回值应如何传递”等细节。在后续的“目标代码生成”章节中，我们会遇到用于为“函数调用生成汇编代码”的 EmitCall()，到时我们再进一步展开讨论。图 5.30 第 34 至 38 行是与函数调用“(GetData(&t0));”有关的汇编代码。

接下来，我们来看一下“删除无用临时变量”的函数 EliminateCode()，如图 5.24 第 17 至 50 行所示，第 25 至 30 行用于删除形如“t4:f();”中的无用临时变量 t4，而第 31 至 41 行用于删除形如“t2:~a;”的无用临时变量 t2，由于表达式“~a”没有副作用，不仅可删去临时变量 t2，同时还可在第 38 至 39 行删去整条中间代码。而函数调用 f() 有副作用，我们就只能在第 29 行删去临时变量 t4。

```

1 void Optimize(FunctionSymbol fsym) {
2     BBlock bb; bb = fsym->entryBB;
3     while (bb != NULL) {
4         PeepHole(bb); bb = bb->next;
5     }
6     bb = fsym->entryBB;
7     while (bb != NULL) {
8         EliminateCode(bb);
9         ExamineJump(bb);
10        bb = bb->next;
11    }
12    bb = fsym->entryBB;
13    while (bb != NULL) {
14        bb = TryMergeBBlock(bb, bb->next);
15    }
16 }
17 static void EliminateCode(BBlock bb) {
18     IRInst inst = bb->insth.next;
19     IRInst ninst; int found = 0;
20     Symbol *opds;
21     find_unused_temp:
22     while (inst != &bb->insth) {
23         ninst = inst->next;
24         opds = inst->opds;
25         if (inst->opcode == CALL) { // t4:f();
26             if (opds[0] && opds[0]->kind == SK_Temp
27                 && opds[0]->ref == 1
28                 && !IsRecordType(opds[0]->ty)) {
29                 opds[0]->ref = 0; opds[0] = NULL;
30             }
31         } else if (inst->opcode <= BCOM
32             || (inst->opcode >= ADDR
33                 && inst->opcode <= MOV)) { // t2:~a;
34             if (opds[0]->kind == SK_Temp
35                 && opds[0]->ref == 1) {
36                 opds[0]->ref = 0; opds[1]->ref--;
37                 if (opds[2]) opds[2]->ref--;
38                 inst->prev->next = inst->next;
39                 inst->next->prev = inst->prev;
40                 found = 1; bb->ninst--;
41             }
}

```

```

42      }
43      inst = inst->next;
44  }
45  if (found) {
46      found = 0;      inst = bb->insth.next;
47      goto find_unused_temp;
48  }
49  return;
50 }

```

图 5.31 Optimize()

函数 Optimize()的代码图 5.31 第 1 至 16 行所示，第 4 行调用的 PeepHole()会对基本块进行窥孔优化，我们已在前面的章节中介绍过这个函数，第 8 行调用的 EliminateCode()函数用于删除无用的临时变量，第 9 行调用的 ExamineJump()函数用于优化跳转语句的跳转目标，而第 14 行调用的 TryMergeBBlock()则尝试进行相邻基本块之间的合并。

图 5.32 给出了函数 ExamineJump()的代码，当我们面对形如第 2 至 10 行的中间代码时，第 4 行的跳转语句是基本块 BB 的最末一条指令，其跳转目标为 BB1，而基本块 BB1 又只有一条跳往 BB2 的无条件跳转指令，此时我们可把基本块 BB 最末一条指令的跳转目标优化为 BB2，如第 14 行所示。图 5.32 第 26 行用于获取基本块 BB 的最后一条指令，当该指令不是跳转指令时，我们从第 29 行直接返回。对基本块 BB 而言，当我们第 4 行的跳转目标 BB1 改为第 14 行的 BB2 后，基本块 BB、BB1 和 BB2 之间的“前驱与后继的关系”发生了变化，我们要从 BB 基本块的后继链表中找到 BB1，将该链表元素改为 BB2，第 31 至 37 行的 do 语句用于在后继链表 bb->succs 中查找 BB1，之后在第 42 行将其改为 BB2，这样 BB2 就成了 BB 基本块的后继结点。而第 43 行的代码用于从 BB1 基本块的前驱链表中删去 BB 基本块，第 44 行用于在基本块 BB2 的前驱链表中添加基本块 BB。第 47 行实现了将 BB 基本块最末一条指令的跳转目标从 BB1 改为 BB2 的优化操作。

```

1  ****
2  BB:
3  ....
4  goto BB1; (或者 if(expr) goto BB1)
5  ...
6  BB1:
7  goto BB2;
8  ...
9  BB2:
10 ...
11 -----可优化为-----
12 BB:
13 ....
14 goto BB2; (或者 if(expr) goto BB2)
15 ...
16 BB1:
17 goto BB2;
18 ...
19 BB2:
20 ...
21 ****
22 void ExamineJump(BBlock bb) {
23     IRInst lasti;
24     CFGEdge succ;
25     BBlock bb1, bb2;

```

```

26 lasti = bb->insth.prev;
27 if (! (lasti->opcode >= JZ
28     && lasti->opcode <= JMP)) {
29     return;
30 }
31 succ = bb->succs;
32 do{
33     if (succ->bb == (BBlock)lasti->opds[0]){
34         break;
35     }
36     succ = succ->next;
37 } while (succ != NULL);
38 bb1 = succ->bb;
39 if (bb1->ninst == 1
40     && bb1->insth.prev->opcode == JMP){
41     bb2 = bb1->succs->bb;
42     succ->bb = bb2;
43     RemovePredecessor(bb1, bb);
44     AddPredecessor(bb2, bb);
45     bb1->ref--;
46     bb2->ref++;
47     lasti->opds[0] = (Symbol)bb2;
48 }
49 }

```

图 5.32 ExamineJump()

在下一小节中，我们会对图 5.31 第 14 行调用的 TryMergeBBlock() 函数进行分析，该函数用于基本块的合并。

## 5.4.2 基本块的合并

我们在第 5.1 节时给出了由基本块构成的双向链表和控制流图，在这一小节中，我们试图把双向链表中相邻的基本块进行合并，当然这种合并需要满足一定条件，同时要保持程序的原有语义。在合并后，控制流图中的前驱与后继关系也要进行调整。我们需要改动的数据结构有图 5.4 中的双向链表和控制流图。需要注意的是，虽然基本块 BB5 和 BB6 在双向链表中相邻，但控制流却不会由 BB5 流入 BB6，双向链表只是维持各基本块在中间代码里的先后顺序，控制流图中的有向边才真正代表了控制流的流向。在本小节中，如未特别声明，“前驱和后继”是针对控制流图而言。

UCC 编译器在中间代码生成时，会产生一些不包含任何中间代码的基本块，如图 5.33 第 22 至 24 行所示，其中的基本块 BB4 和 BB5 中都没有中间代码，这相当于控制流可以从 BB4 流入 BB5，然后再流入 BB6，这也意味着 BB4 是 BB5 的前驱，而 BB5 又是 BB6 的前驱。我们可以将这两个基本块与 BB6 合并。同时，还要把第 20 行的跳转指令改为第 37 行的跳转指令，此时第 35 行的基本块 BB2 是第 38 行基本块 BB6 的唯一前驱，我们可把 BB2 和 BB6 再进行合并，删去第 37 行的无用跳转语句，最终优化后的中间代码如图 5.33 第 45 至 52 行所示。

```

1 int a,b;
2 int main(int argc,char * argv[]){
3     switch(a){
4         case 1:
5         case 2:
6             case 3:
7                 b =300;
8                 break;
9             }
10    b = 500;

```

```

11  return 0;
12 }
13 ///////////////////////////////////////////////////////////////////
14 function main
15 if (a < 1) goto BB8;
16 BB1:
17 if (a > 3) goto BB8;
18 BB2:
19 t0 : a - 1;
20 goto (BB4,BB5,BB6,) [t0];
21 BB3:
22 BB4: // case 1
23 BB5: // case 2
24 BB6: // case 3
25 b = 300;
26 goto BB8;
27 BB8:
28 b = 500;
29 .....
30 ///////////////////////////////////////////////////////////////////
31 function main
32 if (a < 1) goto BB8;
33 BB1:
34 if (a > 3) goto BB8;
35 BB2:
36 t0 : a - 1;
37 goto (BB6,BB6,BB6,) [t0];
38 BB6:
39 b = 300;
40 goto BB8;
41 BB8:
42 b = 500;
43 ///////////////////////////////////////////////////////////////////
44 function main
45 if (a < 1) goto BB3;
46 BB1:
47 if (a > 3) goto BB3;
48 BB2:
49 t0 : a - 1;
50 b = 300;
51 BB8:
52 b = 500;

```

图 5.33 无代码的空基本块

UCC 编译器的 TryMergeBBlock() 函数会对双向链表的相邻基本块进行判断，当满足以下 5 种情况时则进行合并操作，如图 5.34 所示。对于某个基本块 bb 而言，我们用 bb->next 来表示在静态结构中紧随 bb 之后的基本块，例如对图 5.33 第 22 行的 BB4 来说，BB4->next 即为 BB5。

(1) 在图 5.34 情况 1 中，bb2 是基本块 bb1 的唯一后继，而 bb2 也只有一个前驱，此时我们可以把 bb1 和 bb2 进行合并。但有一个特殊情况要处理，即我们要删去如图 5.33 第 37 行的间接跳转指令 IJMP。

(2) 在图 5.34 情况 2 中，基本块 bb1 没有中间代码，此时可删去基本块 bb1，bb1 也不再是 bb2 的前驱。同时，还要修改 bb1 所有前驱的后继链表，将其中的 bb1 改为 bb2，bb1 的所有前驱要成为 bb2 的前驱。

(3) 在图 5.34 情况 3 中，基本块 bb2 没有中间代码，且无前驱，此时可删去 bb2。

(4) 在图 5.34 情况 4 中，基本块 bb2 没有中间代码，其唯一前驱为 bb1，由于 bb2 没有中间代码，控制流一旦进入 bb2，则必然可进入 bb2->next，此时 bb2->next 是 bb2 的后继。我们可删去 bb2，还要使 bb2->next 成为 bb1 的后继。

(5) 在图 5.34 情况 5 中，基本块 bb1 的最末一条指令为 “jump bb2;”，且其中的 bb2 在静态结构中是紧随 bb1 之后的基本块，即 bb2 就是 bb1->next，此时我们可以删去基本块 bb1 中位于最末尾的无条件跳转指令 “jump bb2”。

此处，前驱与后继是对控制流图而言，而非双向链表

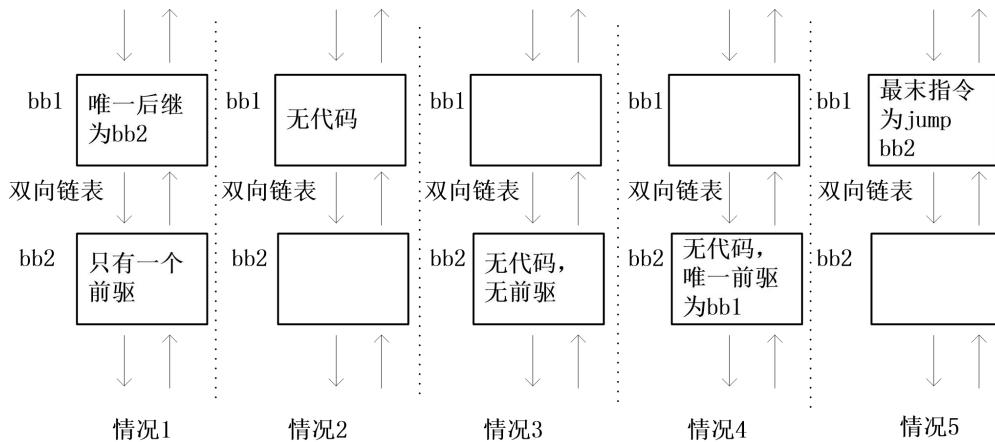


图 5.34 基本块合并的 5 种情况

有了上面的基础后，就不难理解函数 TryMergeBBlock()，其主要代码如图 5.35 所示，第 5 至 34 行对应“情况 1”，第 35 至 36 行对应“情况 2”，第 37 至 39 行对应“情况 3”，第 41 至 44 行对应“情况 4”，而第 46 至 49 行对应“情况 5”。我们只给了“情况 1”的代码，省略了其他情况的代码。

```

1 BBlock TryMergeBBlock(
2     BBlock bb1, BBlock bb2) {
3     if (bb2 == NULL)
4         return bb2;
5     if (bb1->nsucc == 1
6         && bb2->npred == 1
7         && bb1->succs->bb == bb2) {
8         //情况 1
9         CFGEdge succ;
10        if(bb2->ref > 1){
11            IRInst lasti = bb1->insts.prev;
12            if (lasti->opcode == IJMP ){
13                lasti->prev->next = lasti->next;
14                lasti->next->prev = lasti->prev;
15                bb1->ninst--;
16            }
17        }
18        //修改控制流图
19        succ = bb2->succs;
20        while (succ != NULL){
21            RemovePredecessor(succ->bb, bb2);
22            AddPredecessor(succ->bb, bb1);
23            succ = succ->next;
24        }
25        bb1->succs = bb2->succs;
26        bb1->nsucc = bb2->nsucc;
27        //合并指令
28        MergeInstructions(bb1, bb2);
29        //修改双向链表
30        bb1->next = bb2->next;
31        if (bb2->next){
32            bb2->next->prev = bb1;

```

```

33      }
34      return bb1;
35 }else if (bb1->ninst == 0){
36     //情况 2
37 }else if (bb2->ninst == 0
38     && bb2->npred == 0) {
39     //情况 3
40 }
41 else if (bb2->ninst == 0
42     && bb2->npred == 1
43     && bb2->preds->bb == bb1) {
44     //情况 4
45 } else{
46     IRIInst lasti = bb1->insth.prev;
47     if (lasti->opcode == JMP
48         && bb1->succs->bb == bb1->next) {
49         //情况 5
50     }
51 }
52 ....
53 }

```

图 5.35 TryMergeBBlock()

图 5.35 第 10 至 17 行用于删除形如图 5.33 第 37 行的指令“`goto (BB6,BB6,BB6,)[t0];`”，基本块 BB6 在该指令中被多次重复引用，因此第 10 行的 if 条件会成立。当我们把满足图 5.33 “情况 1”的基本块 bb1 和 bb2 进行合并时，由于前驱和后继关系发生变化，我们要通过图 5.35 第 19 至 26 行来修改控制流图，在第 28 行调用函数 `MergeInstructions()` 来合并这两个基本块里的中间代码，还需要在第 29 至 33 行修改双向链表。

至此，我们完成了 UCC 编译器“中间代码生成与优化”的讨论，UCC 编译器可以通过 `ucluiasm.c` 中的函数 `DAssemTranslationUnit()`，把优化后的中间代码打印出来，该函数并不复杂，我们就从略，在下一章中，我们要开始讨论“目标代码生成”，UCC 编译器的目标代码即为 32 位的 x86 汇编代码。

## 5.5 本章习题

1. 请分析以下 C 程序，仿照 UCC 编译器对 `switch` 语句的翻译方法，手工写出以下 `switch` 语句对应的中间代码。

```

int a = 3; int no;
int main(int argc, char * argv[]){
    switch(a){
        case 1: no = 1; break;
        case 2: no = 2; break;
        case 50: no = 3; break;
        case 51: no = 4; break;
        case 60: no = 5; break;
        case 61: no = 6; break;
    }
    return 0;
}

```

2. 在中间代码生成时，UCC 编译器如何处理以下全局数组 `arr1` 和局部数组 `arr2` 的初始化？

```
int arr[8]={10,20};  
void f(void){  
    int arr2[8] = {30,40,50};  
}
```

3. 请上机运行以下程序，结合图 5.15 和图 5.20 分析由 UCC 编译器生成的中间代码。

```
#include <stdio.h>  
struct Data{  
    int offset:12;  
    int pgNo:10;  
    int dirNo:10;  
}dt;  
struct Data * ptr = &dt;  
int main(int argc, char * argv[]){  
    ptr->pgNo++;  
    printf("%d \n",dt.pgNo);  
    return 0;  
}
```

# 第 6 章 汇编代码生成

## 6.1 汇编代码生成简介

历经词法分析、语法分析、语义检查和中间代码生成阶段，我们终于来到了“目标代码生成阶段”，由于 UCC 编译器的目标代码即为 32 位 x86 汇编代码，因此我们就把本章称为“汇编代码生成”。UCC 编译器中的大部分源代码都适用于 Windows 和 Linux 平台，但 Windows 平台上缺省的汇编器支持 Intel 风格的 x86 汇编代码，而 Linux 平台默认的汇编器则采用 AT&T 风格的 x86 汇编代码，两者在汇编语法上有一些差别，为节省篇幅，我们主要针对 Linux 平台的 AT&T 汇编进行讨论。到了这一章，UCC 编译器面对的输入早已不再是 C 源代码，而主要是由各个基本块构成的链表，我们要把基本块里的中间代码翻译成 x86 汇编代码。在第 1.5 节时已介绍过 x86 汇编代码的相关语法和语义，这里不再重复。

还是通过一个简单的例子来了解一下 UCC 编译器的汇编代码生成，如图 6.1 所示，第 1 至 6 行是一个简单的 C 程序，第 11 至 58 行是由 UCC 编译器产生的汇编代码。形如第 13 行 C++ 风格的注释是我们人为加上去的，用于说明第 14 行的汇编代码“.data”是由 Segment() 函数产生。我们省略了 main() 函数对应的汇编代码。

```

1 int a = 10;
2 int f(int num){
3     double b;    b = 1.0;
4     printf("b = %f \n",b);
5     return num+2;
6 }
7 int main(int argc,char*argv[]){
8     a = f(30);      return 0;
9 }
10 ////////////////汇编代码/////////
11 //BeginProgram()
12 #Code auto-generated by UCC
13 // Segment(DATA);
14 .data          //全局静态数据区
15 // EmitStrings();
16 .str0:   .string "b = %f \012"
17 .align 8
18 // EmitFloatConstants()
19 .flt0:   .long    0
20           .long    1072693248
21 // EmitGlobals()
22 .globl  a
23 a:    .long    10
24 // Segment(CODE)
25 .text          //代码区
26 // EmitFunction()
27 .globl  f
28 f:
29 // EmitPrologue()

30 pushl %ebp
31 pushl %ebx
32 pushl %esi
33 pushl %edi
34 movl %esp, %ebp
35 subl $16, %esp
36 .BB0:
37 // EmitBBlock()
38 // b = 1.0;
39 fldl .flt0
40 fstpl -8(%ebp)
41 // printf("b = %f \n",b);
42 leal .str0, %eax
43 subl $8, %esp
44 fldl -8(%ebp)
45 fstpl (%esp)
46 pushl %eax
47 call printf
48 addl $12, %esp
49 // return num+2;
50 movl 20(%ebp), %eax
51 addl $2, %eax
52 // EmitEpilogue
53 movl %ebp, %esp
54 popl %edi
55 popl %esi
56 popl %ebx
57 popl %ebp
58 ret

```

图 6.1 汇编代码生成的例子

在 C 程序中出现的字符串可被视为字符数组(当然对 C 程序员而言，该字符数组的名称不可见)，通过第 15 行的 EmitStrings()函数，UCC 编译器可以为字符串产生形如第 16 行的字符数组，而通过第 18 行的 EmitFloatConstants()函数，则可以为形如“1.0”的浮点常数分配存储空间。UCC 编译器会隐式地为字符串和浮点常数取名，例如第 16 行的“.str0”和第 19 行的“.flt0”，这些名字都以“.”开始。按 C 语言的语法，由 C 程序员命名的变量名或函数名不会以“.”开始，这就可保证不会发生名称上的重名。在汇编代码中，符号名是允许用“.”开始的。第 21 行的 EmitGlobals()函数用于处理 C 程序员定义的全局变量，这会产生形如第 22 至 23 行的汇编代码。第 26 行的 EmitFunction()用于产生函数 f 对应的汇编代码，如图第 27 至 58 行所示。

图 6.1 第 30 至 34 行的代码用于保存寄存器的值，第 35 行用于在栈空间中预留内存空间，用来存放局部变量和临时变量，这部分工作被称为“序言 (Prologue)”，即在函数开始执行时要处理的工作。而图 6.1 第 53 至 57 行被称为“尾声 (Epilogue)”，用于恢复原先保存的寄存器值，第 58 行的汇编指令 ret 用于从栈中取出返回地址并返回。而函数 f 的返回值在第 50 至 51 行进行计算，并保存于寄存器 eax。第 37 行的 EmitBlock()函数用来为某一基本块生成汇编代码。UCC 编译器内部用英文单词 generate 来表示中间代码的生成，而用 emit 来表示汇编代码的生成，这里我们统一译为“生成”。

我们还注意到，图 6.1 第 2 行的形参 num 在汇编代码中，对应的名称是第 50 行的“20(%ebp)”，而第 3 行的局部变量 b 对应的是第 40 行的“-8(%ebp)”，这再次提醒我们，在汇编层次，我们需要为局部变量和形式参数设置新的名称。为了得到某个符号对象 p 在汇编代码中的名称，UCC 编译器定义了函数 GetAccessName()，其接口如下所示，稍后，我们会对这个函数进行分析。

```
static char* GetAccessName(Symbol p);
```

下面，我们来分析一下用于生成图 6.1 汇编代码的函数 EmitTranslationUnit()，如图 6.2 所示，第 1 至 11 行的函数 EmitTranslationUnit()会为整个翻译单元产生汇编代码，我们已在图 6.1 的注释中标出 BeginProgram()等函数的作用。第 8 行的函数 ImportFunctions()用于在 Windows 平台 Intel 风格的汇编中，产生形如“EXTRN f:NEAR32”的函数声明，在 Linux 平台上 ImportFunctions()并无作用。图 6.2 第 12 至 26 行为 EmitFunctions()的代码，通过第 16 行的 while 循环，我们在第 21 行调用 EmitFunction()来为各个函数产生汇编代码。

```

1 void EmitTranslationUnit(
2     AstTranslationUnit transUnit){
3     .....
4     SwitchTableNum = 1;
5     BeginProgram(); Segment(DATA);
6     EmitStrings(); EmitFloatConstants();
7     EmitGlobals(); Segment(CODE);
8     ImportFunctions();
9     EmitFunctions(transUnit);
10    EndProgram(); fclose(ASMFile);
11 }
12 static void EmitFunctions(
13     AstTranslationUnit transUnit){
14     AstNode p; FunctionSymbol fsym;
15     p = transUnit->extDecls;
16     while (p != NULL){
17         if (p->kind == NK_Function){
18             fsym = ((AstFunction)p)->fsym;
19             if (fsym->sclass != TK_STATIC
20                 || fsym->ref > 0) {

```

```

21             EmitFunction(fsym);
22         }
23     }
24     p = p->next;
25 }
26 }
27 void EmitFunction(FunctionSymbol fsym) {
28     BBlock bb;
29     Type rty;
30     int stksize;
31     FSYM = fsym;
32     if (fsym->sclass != TK_STATIC) {
33         Export((Symbol)fsym);
34     }
35     DefineLabel((Symbol)fsym);
36     rty = fsym->ty->bty;
37     if (IsRecordType(rty)
38         && IsNormalRecord(rty)) {
39         VariableSymbol p;
40         CALLOC(p); p->kind = SK_Variable;
41         p->name = "recvaddr";
42         p->ty = T(POINTER);
43         p->level = 1;
44         p->sclass = TK_AUTO;
45         p->next = fsym->params;
46         fsym->params = (Symbol)p;
47     }
48     stksize = LayoutFrame(fsym, PRESERVE_REGS + 1);
49
50     EmitPrologue(stksize);
51     bb = fsym->entryBB;
52     while (bb != NULL) {
53         DefineLabel(bb->sym);
54         EmitBBlock(bb); bb = bb->next;
55     }
56     EmitEpilogue(stksize);
57     PutString("\n");
58 }
```

图 6.2 EmitTranslationUnit()

图 6.2 第 27 至 58 行为函数 EmitFunction() 的代码，在第 33 行调用的 Export() 函数用于在 Linux 平台上产生形如 “.globl f” 的函数声明，第 35 行的 DefineLabel() 函数用于产生形如 “f:” 的标号。按 C 标准的规定，当 “函数的返回值是结构体对象，且该对象的大小不落在 {1, 2, 4, 8}” 时，C 编译器会隐式地为该函数添加一个参数，该参数的类型是指向结构体对象的指针。例如，以下结构体 struct Data 的对象要占 32 字节，C 编译器会为函数 GetData() 隐式地添加一个 struct Data \* recvaddr 参数，图 6.2 第 37 至 47 行的 if 语句用于对此进行处理。

```

struct Data {int dt[8];};
// C 程序员设定的函数接口
struct Data GetData(int num);
// 被 C 编译器隐式地改为
void GetData(struct Data * recvaddr, int num);
```

图 6.2 第 48 行调用的 LayoutFrame() 函数用来计算“形式参数、局部变量和临时变量”在活动记录中的偏移，并返回“局部变量和临时变量”在栈中所占内存的总和，我们会在稍后对这个函数进行分析。图 6.2 第 50 行调用 EmitPrologue() 来产生“序言”，如图 6.1 第 29 至 35 行所示，图 6.1 第 35 行“subl \$16,%esp”中的常数 16，就是“函数 f 里的局部变量和临时变量所占栈内存的总和”。通过图 6.2 第 52 至 55 行的 while 循环，我们可为各基本块产生汇编代码，这主要是由第 54 行调用的 EmitBlock() 函数来完成。图 6.2 第 56 行调用的 EmitEpilogue() 函数来产生“尾声”部分，如图 6.1 第 52 至 58 行所示。

图 6.2 中调用的 EmitStrings() 和 EmitFloatConstants() 等函数并不复杂，我们就不再展开讨论，在后续的章节中，我们把分析的焦点放在基本块的翻译上，即 EmitBlock() 函数。在本节中，我们再来分析一下前面遇到的 LayoutFrame() 和 GetAccessName() 这两个函数，其中 LayoutFrame() 函数的代码如图 6.3 所示。按照 C 标准的规定，在被调函数返回后，寄存器 ebx、esi 和 edi 的值要和函数调用前的值一样，这些寄存器被称为“保值寄存器”。UCC 编译器会在“函数的序言”中把这几个寄存器的值入栈；而在“函数的尾声”中恢复这几个寄存器的值，从而实现“保值”的要求。另外，当被调函数返回时，寄存器 ebp 需要再次指向主调函数的活动记录，因此被调函数也要在栈中先保存 ebp 寄存器的值，因此总共需要由被调函数保护的寄存器有 4 个，即图 6.3 第 16 行的宏 PRESERVE\_REGS 所对应的值。

```

1  ****
2  function(parameter1, parameter2, ....)
3  栈空间示意图:
4  .....
5  parameter2
6  parameter1      20(%ebp)
7  return address   16(%ebp)
8  ebp              12(%ebp)
9  ebx              8(%ebp)
10 esi             4(%ebp)
11 edi             0(%ebp)
12 局部变量与临时变量 -4(%ebp)
13 .....
14 ****
15 // LayoutFrame(fsym, PRESERVE_REGS + 1);
16 #define PRESERVE_REGS 4
17 static int LayoutFrame(
18 FunctionSymbol fsym, int fstParamPos){
19 Symbol p;      int offset;
20 offset = fstParamPos * STACK_ALIGN_SIZE;
21 //计算形式参数的偏移,
22 //从偏移 20 开始递增,
23 //符号名为"20(%ebp)", "24(%ebp)"
24 p = fsym->params;
25 while (p){
26     AsVar(p)->offset = offset;
27     if(p->ty->size == 0){
28         offset += ALIGN(
29             EMPTY_OBJECT_SIZE, STACK_ALIGN_SIZE);
30     }else{
31         offset += ALIGN(
32             p->ty->size, STACK_ALIGN_SIZE);
33     }
34     p = p->next;

```

```

35 }
36 //计算局部变量和临时变量的偏移,
37 //从偏移-4 开始递减,
38 //符号 20(%ebp)
39 offset = 0;      p = fsym->locals;
40 while (p) {
41     if (p->ref == 0) {
42         goto next;
43     }
44     if(p->ty->size == 0){
45         offset += ALIGN(
46             EMPTY_OBJECT_SIZE, STACK_ALIGN_SIZE);
47     }else{
48         offset += ALIGN(
49             p->ty->size, STACK_ALIGN_SIZE);
50     }
51     AsVar(p)->offset = -offset;
52 next:
53     p = p->next;
54 }
55 //返回局部变量和临时变量所占总和
56 return offset;
57 }

```

图 6.3 LayoutFrame()

图 6.3 第 3 至 13 行给出了栈的布局图, 第 1 个形参的位置为“20(%ebp)”, 而第 1 个局部变量或临时变量的位置为“-4(%ebp)”。第 24 至 35 行的 while 循环用于为各个形参计算其偏移, 其偏移从 20 开始依次递增; 第 39 至 54 行的 while 循环用于为局部变量和临时变量计算偏移, 其偏移从“-4”开始依次递减。第 56 行返回“局部变量和临时变量所占用的栈空间的总和”。

接下来, 我们来看一下 Linux 版的函数 GetAccessName(), 其代码如图 6.4 所示, 第 6 至 8 行用于处理整型常数, 在 AT&T 汇编指令中其符号形如“\$4”; 浮点常数对应的名称, 已在图 6.1 第 18 行的 EmitFloatConstants() 函数中被设置为形如“.flt0”; 第 9 至 12 行用来产生形如“.str0”的字符串名和“.BB2”标号。全局变量名和“处于函数体外的静态变量名”, 仍然可以在汇编代码中出现, 第 17 行的赋值语句对此进行设置。为了避免重名, UCC 编译器会对“位于函数体内的静态变量名”进行改名, 得到的名称形如第 23 行的“c.1”, 第 18 至 26 行对此进行处理。

```

1 static char* GetAccessName(Symbol p) {
2     if (p->aname != NULL) {
3         return p->aname;
4     }
5     switch (p->kind) {
6     case SK_Constant:    //movl $4, -4(%ebp)
7         p->aname = FormatName("$%s", p->name);
8         break;
9     case SK_String:       //.str0:    .string "%d \012"
10    case SK_Label:        //.BB2
11        p->aname = FormatName(".%s", p->name);
12        break;
13    case SK_Variable:   //有名变量
14    case SK_Temp:        //临时变量
15        if (p->level == 0 || p->sclass == TK_EXTERN) {

```

```

16          //全局变量或位于函数体外的静态变量
17          p->aname = p->name;
18      } else if (p->sclass == TK_STATIC) {
19          /**处于函数体内的 static 变量*****
20          void f(void) {
21              static int c = 5;
22          }
23          c.1:     .long    5
24          *****/
25          p->aname = FormatName(
26              "%s.%d", p->name, TempNum++);
27      }else{//局部变量、临时变量、形式参数
28          // movl 20(%ebp), %eax
29          p->aname = FormatName("%d(%%ebp)", 
30                          AsVar(p)->offset);
31      }
32      break;
33  case SK_Function: //函数名,f
34      p->aname = p->name;      break;
35  case SK_Offset:   //数组元素, 结构体成员
36  {
37      Symbol base = p->link;
38      int n = AsVar(p)->offset;
39      *****
40      static int arr[4];
41      arr[3] = 200;
42      //movl $200, arr+12
43      *****/
44      if (base->level == 0
45          || base->sclass == TK_STATIC
46          || base->sclass == TK_EXTERN) {
47          p->aname = FormatName("%s%s%d",
48                          GetAccessName(base),
49                          n >= 0 ? "+" : "", n);
50      }else{
51          n += AsVar(base)->offset;
52          p->aname = FormatName("%d(%%ebp)", n);
53      }
54  }
55  break;
56 }
57 return p->aname;
58 }

```

图 6.4 GetAccessName()

对于局部变量、形式参数和临时变量，在汇编代码中，我们用形如“20(%ebp)”这样的符号来表示，图 6.4 第 27 至 32 行会根据我们在图 6.3 的 LayoutFrame() 函数中计算出来的偏移，来设置相应的符号名。在汇编代码中，函数名仍然可以直接使用，图 6.4 第 33 至 34 行对此进行处理。对于全局或静态的“数组元素和结构体成员”，我们可用形如第 42 行的符号“arr+12”来表示，而对于局部的数组或结构体对象，则使用形如“20(%ebp)”的符号。与此相关的代码如图 6.4 第 35 至 56 行所示。当我们已经得到某一符号 p 在汇编代码中的名称 p->aname 后，就没有必要再重复计算，第 2 行的 if 语句会对此进行判断。

在这一节中，我们结合图 6.1 的例子，讨论了汇编代码生成的总体流程。在汇编代码生成中，一个很重要的问题就是寄存器的分配。在后面的章节中，我们还会对“为基本块产生汇编代码”的函数 EmitBlock() 进行讨论。

## 6.2 寄存器的管理

在计算机中，CPU 的速度比内存的速度快得多，编译器应尽量有效地利用寄存器资源，减少对内存的不必要的访问，从而提高由编译器生成的汇编代码的运行速度。在中间代码生成阶段，UCC 编译器用临时变量  $t$  来存放形如 “ $t: a+b;$ ” 的公共子表达式的值；到了汇编代码生成时，UCC 编译器会尽可能地把这些公共子表达式的值存放在寄存器，当需要再次重用时，就可以直接由相应的寄存器中得到。不过，CPU 中寄存器的资源是很有限的，在 32 位的 x86 芯片上，汇编程序员可用的寄存器有 {eax, ebx, ecx, edx, esi, edi, esp, ebp}，其中寄存器 esp 一般用于指向栈顶，而 ebp 一般用于指向活动记录的底部。真正可供选择的寄存器就只有 {eax, ebx, ecx, edx, esi, edi} 这 6 个，当公共子表达式的个数比可用寄存器更多时，我们就要把某些寄存器的值回写 (write back) 到“临时变量对应的内存单元”中，以便腾出可用寄存器来存放其他值。当然如果待回写的寄存器中的值已经不再需要，或者从内存载入 CPU 后并没有发生变化，我们就不必浪费时间去做回写操作了。这有点类似操作系统请求分页中的页面置换算法。为了简化寄存器的分配算法，UCC 编译器只为临时变量分配寄存器，这意味着我们希望尽可能地重用形如 “ $t: a+b;$ ” 这样的公共子表达式。我们还是用一个简单的例子来说明一下 UCC 编译器的寄存器分配，如图 6.5 所示，第 2 至 8 行是函数 f 的代码，第 9 至 13 行是函数 g 的代码，在函数 f 第 5 行完成对 s3 的赋值后，我们有意在第 6 至 7 行再次使用公共子表达式(a+b)和(c+d)。第 32 至 46 行是函数 f 对应的汇编代码，而第 48 至 58 行是函数 g 对应的汇编代码。

```

1 int a,b,c,d,s1,s2,s3;
2 void f(void){
3     s1 = (a+b);
4     s2 = (c+d);
5     s3 = (a+b)+(c+d);
6     s1 = (a+b);
7     s2 = (c+d);
8 }
9 void g(void){
10    s1 = (a+b);
11    s2 = (c+d);
12    s3 = (a+b)+(c+d);
13 }
14 ////////////中间代码///////////
15 function f
16    t0 : a + b;
17    s1 = t0;
18    t1 : c + d;
19    s2 = t1;
20    t2 : t0 + t1;
21    s3 = t2;
22    s1 = t0;
23    s2 = t1;
24 function g
25    t0 : a + b;
26    s1 = t0;
27    t1 : c + d;
28    s2 = t1;
29    t2 : t0 + t1;
30    s3 = t2;
31    ////////////汇编代码///////////
32    f:
33    .....
34    subl $12, %esp
35 .BB0:
36    movl a, %eax //s1 = (a+b);
37    addl b, %eax
38    movl %eax, s1
39    movl c, %ecx //s2 = (c+d);
40    addl d, %ecx
41    movl %ecx, s2
42    movl %eax, %edx //s3= (a+b)+(c+d);
43    addl %ecx, %edx
44    movl %edx, s3
45    movl %eax, s1 //s1 = (a+b);
46    movl %ecx, s2 //s2 = (c+d);
47    .....
48 g:
49    subl $12, %esp
50 .BB1:

```

```

51  movl a, %eax //s1 = (a+b);           55  addl d, %ecx
52  addl b, %eax                         56  movl %ecx, s2
53  movl %eax, s1                         57  addl %ecx, %eax//s3 = (a+b)+(c+d);
54  movl c, %ecx //s2 = (c+d);           58  movl %eax, s3

```

图 6.5 寄存器分配的例子

由图 6.5 第 15 至 23 行的中间代码，我们可以发现函数 f 中有 3 个临时变量，每个都用于存放整数，共占用 12 字节的栈空间，第 34 行的“`subl $12,%esp`”用于在栈中开辟 12 字节内存来存放这 3 个临时变量。我们通过第 36 行的“`movl a,%eax`”指令，把全局变量 a 从内存载入寄存器 eax 中，在执行第 37 行的“`addl b,%eax`”指令后，寄存器 eax 中保存的就是临时变量“t0: a+b”的值，之后寄存器 eax 就一直分配给 t0，直到 t0 不再被使用，或者寄存器不够用时。可以发现，在第 42 行和第 45 行，我们都重用了保存于寄存器 eax 中的公共子表达式(a+b)，在第 44 行我们把“(a+b)+(c+d)”的值保存在寄存器 edx 中。由于寄存器 eax 的值没有发生变化，在第 45 行，我们还可重用保存于 eax 中的 a+b 的值。需要注意的是，一条中间代码可能对应若干条的汇编指令，例如图 6.5 第 16 行的“t0:a+b;”就对应图 6.5 第 36 至 37 行的汇编指令。

如果把寄存器分配的范围扩大到有名的变量（即 C 程序员命名的全局、静态和局部变量），还可进一步地减少内存的访问次数。例如，当我们把全局变量 a 读入某个寄存器 R1 后，之后再次需要 a 的值时，可直接由寄存器 R1 得到，不必再去访问内存。但是由于 C 程序员既可通过变量名来访问“有名的变量”，也可以通过变量的地址 addr 来间接访问，其访问方式比较灵活。如果通过\*addr 的形式来访问全局变量 a，我们可能会把 a 的值加载到另一个寄存器 R2 中，对全局变量 a 再进行写操作时，就可能出现寄存器 R1 和 R2 中的内容不一致的问题。为了避免这样的问题，编译器需要做较复杂的分析。而临时变量只由编译器产生，对 C 程序员不可见，通常情况下，UCC 编译器只对临时变量赋值一次，例如图 6.5 第 15 至 29 行的临时变量。不过也有例外，UCC 编译器在处理形如“`a>0?b:c`”的条件表达式时，确实会对临时变量进行多次赋值。

为了简单起见，避免复杂的数据流分析，UCC 编译器只为临时变量分配寄存器。接下来，我们来看一下用于分配寄存器的函数 `GetRegInternal()`，如图 6.6 第 2 至 18 行所示，第 2 行的参数 width 代表所需要寄存器的宽度，可以是 1 字节（对应寄存器 al、cl 和 dl），也可以 2 字节（对应寄存器 ax、cx、dx、bx、si 和 di），还可以是 4 字节的寄存器（对应 eax、ecx、edx、ebx、esi 和 edi）。第 12 行调用 `FindEmptyReg()` 函数来获取还未被分配的寄存器，如果不存在空寄存器，我们就要在第 14 行通过 `SelectSpillReg()` 函数选择一个要回写的寄存器，再通过第 15 行的 `SpillReg()` 函数将该寄存器中保存的值写回内存，这样该寄存器又可再次被分配。第 17 行在变量 `UsedRegs` 中设置相应的标志位，表示第 i 个寄存器已经被使用。UCC 编译器在为一条中间代码生成汇编指令前，都会先把变量 `UsedRegs` 清 0，在稍后分析函数 `EmitBlock()` 时，可以看到这一点。第 1 行的变量 `UsedRegs` 用于记录“已经分配给当前中间代码”的各个寄存器。

```

1 int UsedRegs;
2 static Symbol GetRegInternal(int width) {
3     int i, endr;    Symbol *regs;
4     switch (width) {
5         case 1:// al, cl ,dl
6             endr = EDX; regs = X86ByteRegs; break;
7         case 2:// AX, CX, DX, BX, SI, DI
8             endr = EDI; regs = X86WordRegs; break;
9         case 4:// EAX, ECX, EDX, EBX, ESI, EDI
10        endr = EDI; regs = X86Regs; break;
11    }

```

```

12 i = FindEmptyReg(endr);
13 if (i == NO_REG){
14     i = SelectSpillReg(endr);
15     SpillReg(X86Regs[i]);
16 }
17 UsedRegs |= 1 << i;      return regs[i];
18 }
19 static int FindEmptyReg(int endr){
20 int i;//找到未被分配出去的寄存器
21 for (i = EAX; i <= endr; ++i) {
22     if (X86Regs[i] != NULL
23         && X86Regs[i]->link == NULL
24         && !(1 << i & UsedRegs))
25     return i;
26 }
27 return NO_REG;
28 }
29 static int SelectSpillReg(int endr){
30 Symbol p;  int i;  int reg = NO_REG;
31 int ref, mref = INT_MAX;
32 for (i = EAX; i <= endr; ++i){
33     if (X86Regs[i] == NULL
34         || (1 << i & UsedRegs)){
35         continue;
36     }
37     p = X86Regs[i]->link;      ref = 0;
38     while (p){
39         if (p->needwb && p->ref > 0){
40             ref += p->ref;
41         }
42         p = p->link;
43     }
44     if (ref < mref) {mref = ref;    reg = i;}
45 }
46 assert(reg != NO_REG);  return reg;
47 }
48 void SpillReg(Symbol reg){
49 Symbol p;  p = reg->link;
50 while (p){//回写保存于 reg 的各临时变量
51     p->reg = NULL;
52     if (p->needwb && p->ref > 0){
53         p->needwb = 0;  StoreVar(reg, p);
54     }
55     p = p->link;
56 }
57 reg->link = NULL;
58 }

```

图 6.6 GetRegInternal()

图 6.6 第 19 至 28 行的函数 FindEmptyReg() 用于查找未被分配的寄存器，第 22 行的条件 “X86Regs[i] !=NULL” 会排除 esp 和 ebp 这两个栈寄存器，第 23 行的条件表示寄存器中没有保存临时变量的值，第 24 行的条件表示该寄存器还没有分配给当前中间代码。若找不到空寄存器，则在第 27 行返回 NO\_REG。当所有寄存器都被分配完了，我们需要通过第 29

至 48 行的函数 SelectSpillReg()选择一个要淘汰的寄存器。这很像请求分页系统中的页面置换算法，我们需要按 FIFO 或 LRU 等算法来淘汰一些页面。UCC 编译器会根据“寄存器对应临时变量的引用次数总和”来做选择，回写引用次数最少的寄存器，第 32 至 45 行对此进行处理。第 48 至 58 行的函数 SpillReg()会把保存在寄存器中的各临时变量的值写回内存，这个动作常被称为“寄存器溢出”。图 6.6 第 52 行的“`p->needwb`”不为 0 时，表示临时变量 `p` 在寄存器和内存中的值已经不一致，而“`p->ref > 0`”表示临时变量 `p` 还需要再次被使用，当这两个条件都符合时，我们才会调用 53 行的 `StoreVar()` 函数来产生写内存的指令。当然，在目前版本的 UCC 编译器中，一个寄存器通常只保存一个临时变量的值，因此第 38 行和第 50 行这两个 while 语句的循环体只执行一次。换言之，第 37 行的链表 `X86Regs[i]->link` 和第 49 行的链表 `reg->link` 上的元素个数都不会超过 1 个。第 51 行设置 `p->reg` 为 `NULL`，表示临时变量 `p` 的值不再保存在寄存器中。

为了加快浮点数的运算，Intel 还提供了一个浮点协处理器 X87，X87 提供了由多个浮点寄存器构成的栈，但为了简单起见，UCC 编译器实际上只使用位于栈顶的浮点数寄存器，来保存某个浮点数临时变量。UCC 编译器中的指针变量 `X87Top` 用于指向该临时变量，当 `X87Top` 不为 `NULL` 时，表示相应临时变量的值保存在协处理器 X87 的栈顶。

```
static Symbol X87Top;
```

在此基础上，我们来看一下“为基本块产生汇编代码”的函数 `EmitBlock()`，如图 6.7 所示第 1 至 24 行所示，第 3 行的 while 循环会遍历基本块中的所有中间代码，第 10 行调用 `EmitIRInst()` 函数为当前中间代码 `inst` 产生汇编指令。由图 6.7 第 25 至 29 行可发现，我们还是通过查函数表的套路来实现相应函数的调用，第 27 行的表格 `Emitter` 中存放了形如第 33 行的函数名 `EmitJump`。第 33 至 38 行的 `EmitJump()` 函数用于为无条件跳转指令产生汇编代码。

```

1 static void EmitBBlock(BBlock bb) {
2     IRInst inst = bb->insth.next;
3     while (inst != &bb->insth) {
4         UsedRegs = 0;
5         if( (inst->opcode >= JZ
6             && inst->opcode <= IJMP)
7             || (inst->opcode == CALL)) {
8             SaveX87Top();
9         }
10        EmitIRInst(inst);
11        if (! (inst->opcode >= JZ
12            && inst->opcode <= IJMP)
13            && inst->opcode != CALL) {
14            DST->ref--;
15            if (SRC1 && SRC1->kind != SK_Function)
16                SRC1->ref--;
17            if (SRC2 && SRC2->kind != SK_Function)
18                SRC2->ref--;
19        }
20        inst = inst->next;
21    }
22    ClearRegs();
23    SaveX87Top();
24 }
25 static void EmitIRInst(IRInst inst) {
26     struct irinst instc = *inst;
27     (* Emitter[inst->opcode])(&instc);
```

```

28   return;
29 }
30 #define DST  inst->opds[0]
31 #define SRC1 inst->opds[1]
32 #define SRC2 inst->opds[2]
33 static void EmitJump(IRInst inst) {
34   BBlock p = (BBlock)DST;
35   DST = p->sym;
36   assert(DST->kind == SK_Label);
37   ClearRegs();
38   PutASMCode(X86_JMP, inst->opds);
39 }
40 void ClearRegs(void) {
41   int i;
42   for (i = EAX; i <= EDI; ++i) {
43     if (X86Regs[i]) {
44       SpillReg(X86Regs[i]);
45     }
46   }
47 }
48 static void SaveX87Top(void) {
49   if (X87Top == NULL)
50     return;
51   assert(X87Top->kind == SK_Temp);
52   if (X87Top->needwb && X87Top->ref > 0) {
53     PutASMCode(
54       X86_STF4 + X87TCode - F4, &X87Top);
55   }
56   X87Top = NULL;
57 }

```

图 6.7 EmitBBlock()

虽然我们在中间代码生成阶段，只对同一基本块内的公共子表达式进行重用，但在遇到条件表达式“(a>0?b:c)”时，确实存在“一个基本块内的临时变量，可能会在其他基本块中被使用”的情况，如下所示，临时变量 t0 会在多个基本块中被赋值，当我们通过“goto BB5”离开基本块 BB3 时，我们需要回写临时变量 t0 的值。

```

d = (a>0?b:c)+1;
//////////////////对应的中间代码/////////////////
if (a <= 0) goto BB4;
BB3:
  t0 = b;      //对临时变量 t0 的赋值
  goto BB5;
BB4:
  t0 = c;      //对临时变量 t0 的赋值
BB5:
  t1 : t0 + 1;
  d = t1;

```

因此，当控制流要通过跳转语句离开基本块，或者遇到函数调用时，我们要对寄存器进行回写操作，图 6.7 第 8 行调用的 SaveX87Top() 用于回写 X87 的栈顶寄存器，而对 X86 寄存器的回写操作，我们将推迟到 EmitJump()、EmitBranch()、EmitIndirectJump() 和 EmitCall() 等函数中执行，分别对应“无条件跳转”，“有条件跳转”，“通过跳转表进行跳转”和“函数调用”。按 C 标准的规定，在函数调用时，主调函数只要回写 eax、ecx 和 edx 这 3 个寄存器

即可。而在遇到跳转语句时，我们通过调用 ClearRegs()函数来回写“eax、ebx、ecx、edx、esi 和 edi”这 6 个寄存器，如图 6.7 第 37 行所示。

函数 SaveX87Top()的代码在图 6.7 第 48 至 57 行，第 53 行调用的 PutASMCode()函数会产生浮点数的回写指令，我们会在后续章节对 PutASMCode()函数进行分析，第 56 行置 X87Top 为 NULL，表示已不存在需要回写的临时变量。

我们还发现，在控制流离开上述基本块 BB4 时，我们也要把临时变量 t0 的值写回内存，这可通过图 6.7 第 22 行调用的 ClearRegs()和第 23 行调用的 SaveX87Top()函数来实现。当我们在图 6.7 第 10 行为一条中间代码产生汇编指令后，可在第 14 至 18 行把各操作数的引用计数减 1，这会对“用于选择溢出寄存器”的函数 SelectSpillReg()产生影响。

在后续章节中，我们会对 EmitBranch()、EmitIndirectJump()和 EmitCall()等为中间代码生成汇编指令的函数进行讨论。

## 6.3 中间代码的翻译

### 6.3.1 由中间代码产生汇编指令的主要流程

在这一小节，我们可把关注的焦点放在“如何把某条中间代码翻译成汇编代码”上。UCC 编译器的中间代码是如下所示的四元式，包括运算符和 3 个操作数。

<运算符 opcode, 目的操作数 DST, 源操作数 SRC1, 源操作数 SRC2>

当然有些中间代码只需要用到 opcode 和 DST 就可以了，例如，无条件跳转指令“goto BB2;”就不需要 SRC1 和 SRC2。为了便于汇编代码的生成，UCC 编译器在 ucl\X86Linux.tpl 中定义了许多汇编指令的模板。在 X86 平台上，无条件跳转的汇编指令为 jmp，对应的指令模板如下所示：

```
//无条件跳转
TEMPLATE(X86_JMP, "jmp %0")
```

其中，X86\_JMP 是模板“jmp %0”的编号，通过编号可找到对应的模板。模板中的“%0”充当占位符的作用，代表第 0 个操作数，即目的操作数 DST，在汇编代码生成时，“%0”会被目的操作数 DST 所替代；而“%1”代表第 1 个操作数，即源操作数 SRC1；“%2”代表第 2 个操作数，即源操作数 SRC2。图 6.8 第 29 至 31 行的宏 DST、SRC1 和 SRC2 分别表示中间指令 inst 里的这 3 个操作数。假设跳转的目标是 BB2，则根据模板“jmp %0”，我们可产生汇编指令“jmp BB2”。

下面，我们来举例介绍一下，如何由中间指令的运算符 opcode 出发，找到对应的汇编指令模板号。在中间代码层次，UCC 编译器用 ADD 来表示加法指令，但在汇编层次，整数加法和浮点数加法对应的汇编指令是不一样的，如图 6.8 第 22 至 25 行所示。为了能通过中间代码的运算符 ADD，快速地找到对应的汇编指令模板编号，ucl\X86Linux.tpl 中的各模板是按一定顺序排列的。例如，当我们要进行两个 double 类型浮点数的加法运算时，其中间指令里的运算符 opcode 为 ADD，类型为 DOUBLE，通过第 1 至 15 行的函数 TypeCode()，我们可得到 DOUBLE 对应的类型编码为 F8。以 ADD 和 F8 作为宏参数，通过第 27 行的宏 ASM\_CODE()，我们就可得到对应的模板号 X86\_ADDF8，如图 6.8 第 26 行所示。

```
1 int TypeCode(Type ty) {
2     ****
3     enum{ // see type.h
4         CHAR, UCHAR, SHORT, USHORT,
5         INT, UINT, LONG, ULONG,
6         LONGLONG, ULONGLONG, ENUM,
```

```

7           FLOAT, DOUBLE, LONGDOUBLE,
8           POINTER, VOID, UNION, STRUCT, ARRAY
9       };
10      *****/
11 static int optypes[] = {
12     I1, U1, I2, U2, I4, U4, I4, U4,
13     I4, U4, I4, F4, F8, F8, U4, V, B, B, B};
14 return optypes[ty->categ];
15 }
16 typedef struct irinst{
17     .....
18     Type ty;
19     int opcode;
20     Symbol opds[3];
21 } *IRInst;
22 //加法    ADD
23 //TEMPLATE(X86_ADDI4,      "addl %2, %0")
24 //TEMPLATE(X86_ADDU4,      "addl %2, %0")
25 //TEMPLATE(X86_ADDF4,      "fadds %2")
26 //TEMPLATE(X86_ADDF8,      "faddl %2")
27 #define ASM_CODE(opcode, tcode) \
28         ((opcode << 2) + tcode - I4)
29 #define DST  inst->opds[0]
30 #define SRC1 inst->opds[1]
31 #define SRC2 inst->opds[2]
32 void PutASMCode(int code, Symbol opds[]){
33     char *fmt = ASMTemplate[code];
34     int i;  PutChar('\t');
35     while (*fmt){
36         switch (*fmt){
37             case ';':
38                 PutString("\n\t"); break;
39             case '%':
40                 fmt++;
41                 if (*fmt == '%'){// %eax
42                     PutChar('%');
43                 }else{ // %0, %1, %2
44                     i = *fmt - '0';
45                     if (opds[i]->reg != NULL){
46                         PutString(opds[i]->reg->name);
47                     }else{
48                         PutString(GetAccessName(opds[i]));
49                     }
50                 }
51                 break;
52             default://其他字符则直接输出
53                 PutChar(*fmt);           break;
54         }
55         fmt++;
56     }
57     PutChar('\n');
58 }

```

图 6.8 由中间指令到汇编代码的过程

通过模板号 X86\_ADDF8，我们就可以在图 6.8 第 33 行查表，得到对应的汇编指令模板“faddl %2”。当然在调用第 32 行的函数 PutASMCode()前，我们需要先为中间指令里的操作数分配必要的寄存器。图 6.8 第 35 至 56 行的 while 循环用来处理形如“faddl %2”的汇编指令模板，在第 43 至 50 行，我们会将占位符“%0”、“%1”和“%2”替换为相应的操作数名称。如果操作数的值已经被加载到寄存器中，则在第 46 行输出对应寄存器的名称，形如“%eax”；否则在第 48 行调用我们在第 6.1 节分析过的函数 GetAccessName()，输出操作数的名称，形如“number”或者“20(%ebp)”。第 41 至 42 行处理模板中形如“%%eax”的字符串，在 AT&T 的汇编指令中，寄存器名前要加一个%，由于在 UCC 的汇编指令模板中，符号%已经被当作转义字符，因此用“%%”表示“%”本身。模板中的“%%eax”经第 32 行的 PutASMCode()函数处理后，会得到“%eax”，其中的“eax”会由第 52 至 53 行进行输出。

需要注意的是，虽然 X86Linux.tpl 中的汇编指令模板是有序排列的，但还是有一些中间指令不能用图 6.8 第 27 行的宏 ASM\_CODE()来找到对应的模板编号。UCC 编译器在生成汇编代码时会进行特殊处理，例如用于“自加”的中间指令 INC，我们就通过“X86\_INCI1 + TypeCode(inst->ty)”来找到与其对应的模板编号，而不是用宏 ASM\_CODE()，如下所示。

```
static void EmitInc(IRInst inst) {
    /*****
    *TEMPLETE(X86_INCI1,      "incb %0")
    *TEMPLETE(X86_INCU1,      "incb %0")
    *TEMPLETE(X86_INCI2,      "incw %0")
    *TEMPLETE(X86_INCU2,      "incw %0")
    *TEMPLETE(X86_INCI4,      "incl %0")
    *TEMPLETE(X86_INCU4,      "incl %0")
    *TEMPLETE(X86_INCF4,      "fldl; fadds %0; fstps %0")
    *****/
    PutASMCode(X86_INCI1 + TypeCode(inst->ty), inst->opds);
}
```

在第 6.2 节，我们介绍了用于分配寄存器的函数 GetRegInternal()，在此基础上，我们来进一步分析“为中间指令里的 I4 或 U4 类型操作数分配寄存器”的函数 AllocateReg()，如图 6.9 所示。第 2 行的参数 index 表示要为哪一个操作数分配寄存器，取值范围为{0, 1, 2}，分别对应 DST、SRC1 和 SRC2。UCC 编译器只为临时变量分配寄存器，第 5 行对此进行检查。如果当前操作数是“已经分配过寄存器”的临时变量，则在第 9 行设置标志位，表示相应的寄存器会被用于“当前中间指令的翻译”。如果我们需要为目的操作数 DST 分配寄存器，而源操作数 SRC1 已在寄存器中，并且在当前中间指令之后，源操作数 SRC1 不再被使用，则可以让 DST 直接重用 SRC1 的寄存器，第 12 至 18 行对此进行处理。对于其他情况，我们在第 19 行调用 GetReg()函数分配一个 4 字节寄存器，并在第 21 行通过 MOV 指令把源操作数加载到寄存器中，在第 23 行调用 AddVarToReg()把临时变量 p 添加到寄存器 reg 的链表 reg->link 中。在 UCC 编译器内部，只有把临时变量 p 添加到寄存器 reg 对应的链表 reg->link 时，才意味着我们确实把寄存器 reg “长期” 分配给了临时变量 p。在目前版本的 UCC 中，该链表最多只存放一个临时变量。

```
1 static void AllocateReg(
2             IRInst inst, int index){
3     Symbol reg;     Symbol p;
4     p = inst->opds[index];
5     if (p->kind != SK_Temp) {
6         return;
7     }
8     if (p->reg != NULL) {
```

```

9     UsedRegs |= 1 << p->reg->val.i[0];
10    return;
11 }
12 if (index == 0 && SRC1->ref == 1
13     && SRC1->reg != NULL) {
14     reg = SRC1->reg;
15     reg->link = NULL;
16     AddVarToReg(reg, p);
17     return;
18 }
19 reg = GetReg();
20 if (index != 0) {
21     Move(X86_MOVI4, reg, p);
22 }
23 AddVarToReg(reg, p);
24 }
25 static void AddVarToReg(
26     Symbol reg, Symbol v){
27 assert(v->kind == SK_Temp );
28 assert(GetListLen(reg) == 0);
29 v->link = reg->link;
30 reg->link = v;
31 v->reg = reg;
32 }
33 Symbol GetReg(void){
34     return GetRegInternal(4);
35 }
36 static void ModifyVar(Symbol p){
37     Symbol reg;
38     if (p->reg == NULL) {
39         return;
40     }
41 //p->needwb = 0;
42 //reg = p->reg;
43 //assert(GetListLen(reg) == 1);
44 //assert(p == reg->link);
45 //SpillReg(reg);
46 //AddVarToReg(reg, p);
47     p->needwb = 1;
48 }
49 static void Move(int code,
50     Symbol dst, Symbol src){
51     Symbol opds[2];
52     opds[0] = dst;
53     opds[1] = src;
54     PutASMCode(code, opds);
55 }

```

图 6.9 为中间指令里的 I4 或 U4 类型操作数分配寄存器

函数 AddVarToReg()的代码如图 6.9 第 25 至 32 行所示，第 29 至 30 行实现了链表插入操作，第 31 行用于记录“临时变量是存放在哪个寄存器中”。如果某个变量存放于寄存器中，当我们修改寄存器中的值时，并不马上写回内存，而是设置一个标志位 needwb，表示需要在稍后进行回写，这可通过第 36 行的函数 ModifyVar()来实现。

当操作数为浮点数时，我们需要为之分配浮点寄存器，为了简单起见，UCC 编译器只使用了 X87 的栈顶寄存器。浮点寄存器的管理并不是通过 GetReg()或 AllocateReg()等函数，我们会在汇编代码生成时，对浮点数进行专门处理。当操作数为 I1 或 U1 类型时，我们通过图 6.10 第 53 行的 GetByteReg()来获得单字节寄存器；而当操作数为 I2 或者 U2 类型时，经由图 6.10 第 56 行的 GetWordReg()来获得双字节寄存器。

接下来，我们以 EmitMove()函数为例来进行讨论，该函数所处理的中间指令如下所示，

<MOV, DST, SRC1, NULL>

该中间指令执行 “DST = SRC1;” 的赋值操作，函数 EmitMove()的代码如图 6.10 所示，第 4 至 7 行用于处理浮点数之间的赋值，我们在第 5 行调用 EmitX87Move()函数来产生相应的汇编代码。对于结构体对象之间的赋值，我们通过第 9 行的 EmitMoveBlock()函数来处理。我们会在稍后以这两个函数进行分析。

```

1 static void EmitMove(IRInst inst) {
2     int tcode = TypeCode(inst->ty);
3     Symbol reg;
4     if (tcode == F4 || tcode == F8) {
5         EmitX87Move(inst, tcode);
6         return;
7     }
8     if (tcode == B) {
9         EmitMoveBlock(inst);
10        return;
11    }
12    switch (tcode){// DST = SRC1;
13    case I1: case U1:
14        if (SRC1->kind == SK_Constant) {
15            // "movb %1, %0"
16            Move(X86_MOV11, DST, SRC1);
17        }else{ // a = b;
18            reg = GetByteReg();
19            Move(X86_MOV11, reg, SRC1);
20            Move(X86_MOV11, DST, reg);
21        }
22        break;
23    case I2: case U2:
24        if (SRC1->kind == SK_Constant) {
25            Move(X86_MOV12, DST, SRC1);
26        }else{ // a = b;
27            reg = GetWordReg();
28            Move(X86_MOV12, reg, SRC1);
29            Move(X86_MOV12, DST, reg);
30        }
31        break;
32    case I4: case U4:
33        if (SRC1->kind == SK_Constant) {
34            Move(X86_MOV14, DST, SRC1);
35        }else{
36            AllocateReg(inst, 1);
37            AllocateReg(inst, 0);
38            if (SRC1->reg == NULL
39                && DST->reg == NULL){
40                // a = b;

```

```

41         reg = GetReg();
42         Move(X86_MOVI4, reg, SRC1);
43         Move(X86_MOVI4, DST, reg);
44     }else{
45         Move(X86_MOVI4, DST, SRC1);
46     }
47 }
48     ModifyVar(DST);      break;
49 default:
50     assert(0);
51 }
52 }
53 Symbol GetByteReg(void){
54     return GetRegInternal(1);
55 }
56 Symbol GetWordReg(void){
57     return GetRegInternal(2);
58 }

```

图 6.10 EmitMove()

当 DST 和 SRC1 为 char(或者 unsigned char)时，我们执行图 6.10 第 13 至 22 行的代码。按照 X86 汇编指令的寻址要求，同一条汇编指令的两个操作数不可以都在内存中。因此，当我们面对以下赋值语句“`a = b;`”时，我们要在第 18 行通过 `GetByteReg()` 函数获取一个单字节的寄存器，然后在第 19 行产生 `movb` 指令，把源操作数 `b` 从内存加载到寄存器中，之后在第 20 行产生另一条 `movb` 指令，把寄存器中的值传给目的操作数 `a`，对该寄存器的使用到此就已结束，我们并未把该寄存器“长期”分配给 `a` 或者 `b`。当然，如果源操作数是整数常量，我们在第 16 行产生一条 `movb` 指令即可，例如以下的“`movb $49,a`”。

```

char a, b;
a = b;
a = '1';
//////////对应的汇编代码//////////
movb b, %al // a = b;
movb %al, a
movb $49, a // a = '1';

```

图 6.10 第 32 至 48 行用来处理 I4 或 U4 类型的赋值操作“`DST=SRC1;`”。图 6.10 第 34 行用来处理形如“`x = 2015;`”的赋值，`x` 可以是临时变量 `t` 也可是有名变量 `a`。第 36 至 37 行调用 `AllocateReg()` 函数为 `DST` 和 `SRC1` 分配 4 字节寄存器。而第 38 至 43 行用来处理形如“`a=b;`”有名变量之间的赋值，按 x86 汇编指令的寻址要求，我们可在第 41 行通过 `GetReg()` 函数获取一个寄存器来做中转，从而产生两条 `movl` 指令。第 45 行用于处理形如“`t=x;`”或者“`x=t;`”的赋值，其中 `t` 为临时变量，由于临时变量的值在寄存器中，此时我们产生一条 `movl` 汇编指令即可。当 `DST` 为临时变量时，由于进行了“`t=x;`”的赋值操作，`t` 对应寄存器中的内容会发生变化，我们会在第 48 行调用 `ModifyVar()` 函数来设置回写标志位。

图 6.10 第 23 至 31 行用来处理 I2 或 U2 类型的赋值操作“`DST=SRC1;`”，基本的流程与第 13 至 22 行类似，我们就不再啰嗦。对于形如“`a+b`”或者“`~a`”的整型算术表达式而言，在语义检查时，UCC 会把低于 `int` 型的整型操作数隐式地提升为 `int` 型，也就是说，整型的公共子表达式的类型是 I4 或者 U4，不会是 I1、U1、I2 或者 U2。因此，在图 6.10 中，用于处理 I4 和 U4 的第 32 至 48 行的代码就会比其他情况来得复杂一些。

接下来，我们来分析一下图 6.10 中用到的 `EmitX87Move()` 和 `EmitMoveBlock()` 这两个函数。图 6.11 第 1 至 25 行的函数 `EmitX87Move()` 用于实现 F4 或 F8 浮点型的赋值操作“`DST`

= SRC1;”，当 SRC1 的值不在 x87 栈顶寄存器时，我们通过第 4 行的 SaveX87Top() 把当前栈顶寄存器的值进行回写，然后在第 6 行产生汇编指令，把 SRC1 从内存中加载到 x87 栈顶寄存器。执行到第 8 行时，SRC1 的值已保存在栈顶寄存器中，我们分以下情况来讨论：

(1) DST 不是临时变量，而 SRC1 是临时变量，且在 “DST = SRC1;” 之后 SRC1 还要在其他指令中被使用。我们就在第 11 行把 x87 栈顶寄存器的值传送到 DST，但不把栈顶寄存器弹出，即该寄存器中仍然保存 SRC1 的值；

(2) DST 不是临时变量，而 SRC1 不是临时变量或者在 “DST = SRC1;” 之后 SRC1 不再被使用。我们就在第 17 行把 x87 栈顶寄存器的值传送到 DST，然后把 X87 栈顶寄存器出栈，之后在第 18 行设置 X87Top 为 NULL，表示 x87 栈顶寄存器中没有保存任何临时变量的值。

(3) DST 是临时变量。我们就在第 21 行设置其回写标志位 needwb，在第 22 行把 X87Top 改为 DST，表示临时变量 DST 的值被保存在 x87 栈顶寄存器中。

```

1 static void EmitX87Move(
2     IRIInst inst, int tcode){// DST = SRC1;
3     if (X87Top != SRC1){
4         SaveX87Top();
5         // TEMPLATE(X86_LDF4,      "flds %0")
6         PutASMCode(X86_LDF4 + tcode - F4, &SRC1);
7     }
8     if (DST->kind != SK_Temp){
9         if(SRC1->kind == SK_Temp
10            && IsLiveAfterCurEmit(inst,1)){
11             PutASMCode(X86_STF4_NO_POP
12                         + tcode - F4, &DST);
13             SRC1->needwb = 1;
14             X87Top = SRC1;
15             X87TCode = tcode;// F4 or F8
16         }else{
17             PutASMCode(X86_STF4 + tcode - F4, &DST);
18             X87Top = NULL;
19         }
20     }else{
21         DST->needwb = 1;
22         X87Top = DST;
23         X87TCode = tcode;
24     }
25 }
26 static void EmitMoveBlock(IRIInst inst){
27     if(inst->ty->size == 0){
28         return;
29     }
30     SpillReg(X86Regs[EDI]);
31     SpillReg(X86Regs[ESI]);
32     SpillReg(X86Regs[ECX]);
33     /*****
34     typedef struct{
35         int data[10];
36     }Data;
37     Data a,b;
38     int main(int argc,char * argv[]){
39         a = b;

```

```

40     return 0;
41 }
42 -----
43 leal a, %edi
44 leal b, %esi
45 movl $40, %ecx
46 rep movsb
47 ****
48 SRC2 = IntConstant(inst->ty->size);
49 // TEMPLATE(X86_MOVBL,
50 //     "leal %0, %%edi;""
51 //     "leal %1, %%esi;""
52 //     "movl %2, %%ecx;""
53 //     "rep movsb"
54 PutASMCode(X86_MOVBL, inst->opds);
55 }

```

图 6.11 EmitX87Move 和 EmitMoveBlock

图 6.11 第 26 至 55 行的 EmitMoveBlock() 用于结构体对象之间的赋值，对于第 39 行的 “`a = b;`” 来说，结构体对象 `a` 要占 40 字节，对应的汇编代码在第 43 至 46 行，其中用到了寄存器{`edi, esi, ecx`}，`esi` 用于存放 `b` 的地址，`edi` 用于存放 `a` 的地址，`ecx` 存放要复制的字节个数，第 46 行的 “`rep movsb`” 实现了内存复制。由于用到了{`edi, esi, ecx`} 这几个寄存器，我们需要在第 30 至 32 行通过 SpillReg() 函数进行必要的回写操作。第 48 行根据类型信息，得到要复制的字节个数，第 54 行产生进行内存复制的汇编指令。

### 6.3.2 为算术运算产生汇编代码

在这一小节中，我们要讨论的中间指令形如 “`t1: a+b;`” 或者 “`t2: ~number`”，这些指令用于进行一元或二元算术运算，并把运算结果保存在临时变量 `t1` 或者 `t2` 中，其中间指令的格式如下所示：

```

<运算符 opcode, 目的操作数 DST, 源操作数 SRC1, 源操作数 SRC2>
<ADD, t1, a, b>           // t1: a+b;
<BCOM, t2, number, NULL> // t2: ~number;

```

由于在一条 x86 汇编指令中，最多只允许出现 2 个操作数，而中间指令 “`DST:SRC1+SRC2`” 有 3 个操作数，我们需要产生多条 x86 汇编指令来实现该中间指令。在汇编指令中，整数加法运算对寄存器没什么特别要求，我们可按以下步骤来处理：

(1) 调用 AllocateReg() 函数依次为 `SRC1`、`SRC2` 和 `DST` 分配寄存器。`DST` 是用于保存运算结果的临时变量，必然可分配到一个寄存器，不妨记为 `Rx`。而如果 `SRC1` 和 `SRC2` 不是临时变量，则没有分配到寄存器。

(2) 若 `DST` 和 `SRC1` 对应的寄存器不一样，我们可产生一条 `movl` 指令，把 `SRC1` 的值传送到寄存器 `Rx` 中。若 `DST` 和 `SRC1` 对应同一个寄存器，则不必产生用于加载 `SRC1` 的 `movl` 指令，此时 `SRC1` 的值已经在 `Rx` 中。

(3) 产生加法指令，进行 `SRC2` 和 `Rx` 的加法，并把结果存于寄存器 `Rx` 中。

按这样的思路，我们可为 “`t1 : a+b;`” 产生以下汇编代码：

```

movl a, %eax
addl b, %eax

```

而形如 “`t2: ~number`” 的中间指令只有 `DST` 和 `SRC1` 这两个操作数，在上述第(1)步中，我们就不必为 `SRC2` 分配寄存器，其他的步骤类似，我们可为 “`t2: ~number`” 产生以下汇编

代码：

```
movl num, %eax
notl %eax
```

不过，有些 x86 汇编指令对寄存器有特定的要求，比如整数的乘法运算就要求源操作数 SRC1 的值被加载到寄存器 eax 中。而整数的除法运算，要求源操作数 SRC1 的值被加载到 eax 中。若是有符号数的除法运算，则寄存器 edx 的所有位都被设置为 SRC1 的符号位；如果是无符号数的除法运算，则寄存器 edx 被置为全 0。例如我们可为中间指令“t3: a /b;”产生以下汇编代码，其中 a 和 b 为有符号整数。

```
movl a, %eax      //把 SRC1 加载到 eax
cdq               //把符号位扩展到 edx 寄存器
idivl b          //进行除法运算[edx: eax] / SRC2, 商存于 eax, 余数存于 edx,
                  //此时 eax 中的值就是临时变量 t3 的值
```

在 x86 “左移或右移”的汇编指令中，如果要把左移或右移的位数存放于寄存器中，则必须使用单字节寄存器 cl，如下所示：

```
int a, c; char len = 3;
c = a << len;
////////对应的中间代码/////////
t4 : (int)(char)len;           //把 char 提升为 int
t5 : a << t4;
c = t5;
```

对应的汇编代码如下所示，我们可以看到，在汇编指令“shll %cl, %edx”中，我们是用单字节寄存器 cl 来存放操作数 len 的值。

```
movsb1 len, %eax //t4 : (int)(char)len;
movl %eax, %ecx //t5: a << t4
movl a, %edx
shll %cl, %edx
movl %edx, c     // c = t5;
```

有了这些基础后，我们可以来讨论一下“为算术运算生成汇编代码”的函数 EmitAssign()，如图 6.12 所示。第 47 至 56 行用于处理对寄存器没有特别要求的二元运算，形如

“DST:SRC1+SRC2;”。第 44 至 45 行用于处理形如“DST:~SRC1”的一元运算，此时我们不必为 SRC2 分配寄存器。第 33 至 43 行用于为“左移或右移指令里的 SRC2”分配寄存器 ecx，并在第 39 行把 SRC2 加载寄存器 ecx，之后在第 41 行把 SRC2 改为单字节寄存器 cl，即 4 字节寄存器 ecx 的低 8 位。

```
1 static void EmitAssign(IRInst inst){
2     int code;    int tcode= TypeCode(inst->ty);
3     if (tcode == F4 || tcode == F8){
4         EmitX87Assign(inst, tcode);    return;
5     }
6     code = ASM_CODE(inst->opcode, tcode);
7     switch (code) {
8     case X86_DIVI4: case X86_DIVU4:// DST: SRC1/SRC2
9     case X86_MODI4: case X86_MODU4:// DST: SRC1%SRC2
10    case X86_MULU4: // DST: SRC1*SRC2
11        if (SRC1->reg == X86Regs[EAX]){
12            SRC1->needwb = 0;    SpillReg(X86Regs[EAX]);
13        } else{
14            SpillReg(X86Regs[EAX]);
15            Move(X86_MOVI4, X86Regs[EAX], SRC1);
16        }
17        SpillReg(X86Regs[EDX]);
```

```

18     UsedRegs = 1 << EAX | 1 << EDX;
19     if (SRC2->kind == SK_Constant) {
20         Symbol reg = GetReg();
21         Move(X86_MOVI4, reg, SRC2);
22         SRC2 = reg;
23     }else{
24         AllocateReg(inst, 2);
25     }
26     PutASMCode(code, inst->opds);
27     if (code == X86_MODI4 || code == X86_MODU4) {
28         AddVarToReg(X86Regs[EDX], DST);
29     }else{
30         AddVarToReg(X86Regs[EAX], DST);
31     }
32     break;
33 case X86_LSHI4: case X86_LSHU4: //DST:SRC1<<SRC2
34 case X86_RSHI4: case X86_RSHU4: //DST:SRC1>>SRC2
35     AllocateReg(inst, 1);
36     if (SRC2->kind != SK_Constant) {
37         if (SRC2->reg != X86Regs[ECX]) {
38             SpillReg(X86Regs[ECX]);
39             Move(X86_MOVI4, X86Regs[ECX], SRC2);
40         }// shll %cl, %eax
41         SRC2 = X86ByteRegs[ECX]; UsedRegs = 1 << ECX;
42     }
43     goto put_code;
44 case X86_NEGI4: case X86_NEGU4: // DST: -SRC1
45 case X86_BCOMI4:case X86_BCOMU4:// DST: ~SRC1
46     AllocateReg(inst, 1);      goto put_code;
47 default:    // DST:SRC1+SRC2
48     AllocateReg(inst, 1);      AllocateReg(inst, 2);
49 put_code:
50     AllocateReg(inst, 0);
51     if (DST->reg != SRC1->reg){
52         Move(X86_MOVI4, DST, SRC1);
53     }// "addl %2, %0"
54     PutASMCode(code, inst->opds);      break;
55 }
56 ModifyVar(DST);
57 }

```

图 6.12 EmitAssign()

图 6.12 第 8 至 32 行用于为整数乘法、除法和取余运算产生汇编指令，第 11 至 16 行把 SRC1 的值暂存于寄存器 eax 中，第 17 行把 edx 寄存器的值回写到内存中，我们会在 [edx:eax] 中存放经符号位扩展后的 SRC1。第 19 至 24 行用于为 SRC2 分配必要的寄存器，第 26 行产生汇编指令来进行乘法或除法运算，之后寄存器 eax 中存放的是“乘法运算的结果”或者“除法运算的商”，而寄存器 edx 中存放的是“除法运算的余数”，如第 27 至 31 行所示。

由于浮点数运算的汇编指令与整数不同，我们需要在图 6.12 第 4 行调用 EmitX87Assign() 函数，对浮点数算术运算进行处理。我们先举一个例子来说明，对于形如“DST: SRC1+SRC2”的二元浮点数运算来说，我们可按以下步骤来生成汇编代码：

- (1) 若 SRC1 不在 x87 栈顶寄存器中，则先对 x87 栈顶寄存器进行必要的回写，然后把 SRC1 加载到 x87 栈顶寄存器中。

(2) 对 x87 栈顶寄存器与 SRC2 进行加法运算, 结果存于 x87 栈顶寄存器中。按这个思路, 我们可为浮点数中间指令 “t6: d+e” 产生以下汇编代码:

```
flds d      //把浮点数 d 从内存加载到 x87 栈顶寄存器
fadds e     //完成加法运算后, x87 栈顶寄存器即为 t6 的值
```

而对于形如 “DST: -SRC1” 的一元运算来说, 我们在上述第(2)步中, 只要对 x87 栈顶寄存器进行一元运算即可, 运算后的结果仍存于 x87 栈顶寄存器中。例如, 我们可为 “t7: -d” 产生以下汇编代码。

```
flds d      //把浮点数 d 从内存加载到 x87 栈顶寄存器
fchs       //把符号位取反, x87 栈顶寄存器即为 t7 的值
```

现在, 我们可以来分析一下产生这些浮点运算汇编指令的函数 EmitX87Assign(), 如图 6.13 所示。可分以下几种情况来讨论:

(1) SRC1 还未加载到 x87 栈顶寄存器中。此时, 我们先通过第 5 行进行必要的回写, 然后在第 6 行把 SRC1 从内存加载到 x87 栈顶寄存器中。

(2) SRC1 的值已经在 x87 栈顶寄存器中, 且 SRC1 还要在后续的中间指令中被使用。此时, 我们需要在第 10 行把 SRC1 的值回写到内存中。其原因是, 在浮点运算完成后, x87 栈顶寄存器保存的是 DST 的值而非 SRC1 的值。

(3) SRC1 的值已经在 x87 栈顶寄存器中, 且操作数 SRC2 就是 SRC1。此时, 进行的是二元浮点算术运算, 我们也要在第 10 行把 SRC1 的值回写到内存中。其原因是, 遇到形如第 13 行的模板 “faddl %2” 时, 我们要把 x87 栈顶寄存器与位于内存中的 SRC2 进行加法运算, 而此时临时变量 SRC1 (即 SRC2) 的值可能还不在内存中, 第 10 行会把 x87 栈顶寄存器的值写回内存。

```
1 static void EmitX87Assign(IRInst inst, int tcode) {
2   // DST: SRC1 + SRC2;    二元运算
3   // DST: -SRC1;          一元运算
4   if(SRC1 != X87Top) {
5     SaveX87Top();
6     PutASMCode(X86_LDF4 + tcode - F4, &SRC1);
7   }else{
8     if(SRC1 == SRC2 || IsLiveAfterCurEmit(inst,1)){
9       //把 SRC1 的值回写到内存中, 但不把 x87 栈顶寄存器弹出
10      PutASMCode(X86_STF4_NO_POP + tcode - F4, &SRC1);
11    }
12  }
13 // TEMPLATE(X86_ADDF8,      "faddl %2")
14 // TEMPLATE(X86_NEGF4,      "fchs")
15 PutASMCode(ASM_CODE(inst->opcode, tcode), inst->opds);
16 DST->needwb = 1;
17 X87Top = DST;
18 X87TCode = tcode;
19 }
```

图 6.13 EmitX87Assign()

图 6.13 第 15 行根据相应的模板, 调用 PutASMCode() 函数产生浮点运算的汇编指令, 其运算结果保存在 X87 栈顶寄存器中, 即临时变量 DST 的值。

### 6.3.3 为跳转指令产生汇编代码

在这一小节中, 我们要为 “有条件跳转”、“无条件跳转” 和 “间接跳转” 产生相应的汇编指令, 其中间指令的四元式如下所示:

<运算符 opcode, 目的操作数 DST, 源操作数 SRC1, 源操作数 SRC2>

(1) 有条件跳转, 例如 “if(a <= b) goto BB2;”, 其四元式为:

```
<JLE, BB2, a, b>
//////////对应的汇编代码//////////
movl a, %eax //把 SRC1 的值暂存在寄存器 eax 中
cmpb b, %eax //比较 eax 和 b 的大小
jle .BB2 //进行有条件跳转
```

(2) 无条件跳转, 例如 “goto BB3;”, 其四元式为:

```
<JMP, BB3, NULL, NULL>
//////////对应的汇编代码//////////
jmp .BB3
```

(3) 间接跳转, 例如 “goto (BB4,BB5,BB6)[t0];”, 在翻译 switch 语句时会产生间接跳转指令, 其四元式为:

```
<IJMP, [BB4, BB5, BB6, NULL], t0, NULL>
//////////对应的汇编代码//////////
.data
swtchTable1:    .long    .BB4
                 .long    .BB5
                 .long    .BB6
.text
jmp *swtchTable1(%eax, 4)
```

由于跳转指令位于基本块的最末尾, 我们在图 6.7 的函数 EmitBBlock()中, 已对 x87 栈顶寄存器做过回写操作。在本小节, 我们还要调用 ClearRegs()函数, 对 x86 CPU 中的寄存器进行回写, 如图 6.14 第 7 行、第 26 行和第 54 行所示。控制流即将随着跳转指令的执行而进入另一个基本块, 而 UCC 编译器仅在同一基本块内对公共子表达式进行重用, 因此, 即使跳转指令里的操作数 DST、SRC1 和 SRC2 是临时变量, 我们没有必要为其“长期”分配寄存器, 换言之, 我们不会调用在前面的章节中介绍过的 AllocateReg()函数, 而是调用如图 6.14 第 13 行所示的 PutInReg()函数, 把操作数 SRC1 的值暂存到某个寄存器中。

图 6.14 第 1 至 30 行的 EmitBranch()函数用于为“有条件跳转”产生汇编代码, 当操作数是浮点数时, 我们在第 8 行调用 EmitX87Branch()函数来处理。当操作数是整数时, 我们会在第 11 行做进一步判断。由于常数会以“立即数”的形式存在于代码区中, 当程序运行时, CPU 会从代码区里预读机器指令, 从而把立即数也加载入 CPU, 因此当操作数 SRC2 是常数时, 我们可以不必把 SRC1 的值加载到寄存器中, 这不会违反“同一条 X86 汇编指令的两个操作数不可以都在内存中”的寻址要求。这意味着我们可以生成形如“cmpl \$3, a”的比较指令, 但不可以生成形如“cmpl b, a”的比较指令。图 6.14 第 11 至 14 行会在“SRC2 存在且 SRC2 不是常数时”, 通过第 13 行调用的 PutInReg()函数, 把 SRC1 的值加载到某个寄存器中。第 17 行判断 SRC1 的值是否在寄存器中, 如果已载入寄存器, 我们可在第 18 行把中间指令里的源操作数 SRC1 改为 SRC1->reg, 之后就可生成形如“cmpl b, %eax”的比较指令。第 28 行调用 PutASMCODE()函数来产生比较和跳转指令, 形如“cmpl b, %eax; jle .BB2”。

```
1 static void EmitBranch(IRInst inst) {
2     // <JLE, BB2, a, b> 有条件跳转
3     int tcode = TypeCode(inst->ty);
4     BBlock p = (BBlock)DST;
5     DST = p->sym;
6     if (tcode == F4 || tcode == F8) {
7         ClearRegs();
8         EmitX87Branch(inst, tcode);
9     }
10 }
```

```

10  }
11  if (SRC2) {
12      if (SRC2->kind != SK_Constant) {
13          SRC1 = PutInReg(SRC1);
14      }
15  }
16  SRC1->ref--;
17  if (SRC1->reg != NULL) {
18      SRC1 = SRC1->reg;
19  }
20  if (SRC2) {
21      SRC2->ref--;
22      if (SRC2->reg != NULL) {
23          SRC2 = SRC2->reg;
24      }
25  }
26  ClearRegs();
27 //TEMPLATE(X86_JLEI4,"cmpl %2, %1;jle %0")
28 PutASMCode(ASM_CODE(inst->opcode, tcode)
29             , inst->opds);

30 }
31 static void EmitX87Branch(
32     IRInst inst, int tcode){
33 // <JLE, BB2, a, b> 有条件跳转
34 PutASMCode(X86_LDF4 + tcode - F4, &SRC1);
35 //"flds %2; fucompp;fnstsw %%ax;""
36 //"test $0x5, %%ah; jp %0"
37 PutASMCode(ASM_CODE(inst->opcode, tcode),
38             inst->opds);
39 }
40 static Symbol PutInReg(Symbol p){
41     Symbol reg;
42     if (p->reg != NULL) {
43         assert(p->kind == SK_Temp);
44         return p->reg;
45     }
46     reg = GetReg();
47     Move(X86_MOVI4, reg, p);
48     return reg;
49 }
50 static void EmitJump(IRInst inst){
51 // <JMP, BB3, NULL, NULL> 无条件跳转
52 BBlock p = (BBlock)DST;
53 DST = p->sym;
54 ClearRegs();
55 //TEMPLATE(X86_JMP,"jmp %0")
56 PutASMCode(X86_JMP, inst->opds);
57 }

```

图 6.14 EmitBranch() 和 EmitJump()

图 6.14 第 31 至 39 行的 EmitX87Branch() 用于处理浮点数的有条件跳转，我们会在第 34 行先把操作数 SRC1 加载到 x87 栈顶寄存器中，然后在第 37 行生成浮点数比较和跳转的指

令，形如第 35 至 36 行的模板所示。我们已在第 1.5 节时介绍过这些指令，这里不再重复。图 6.14 第 50 至 56 行的函数 EmitJump()会产生无条件跳转指令。

接下来，我们来讨论一下为“间接跳转”产生汇编代码的函数 EmitIndirectJump()，如图 6.15 所示。图 6.15 第 7 行调用 PutInReg()函数把操作数 SRC1 加载到寄存器中，第 10 行用于产生“.data”，表示接下来的内容为数据区，第 11 至 17 行创建一个名称形如“swtchTable1”的符号对象，第 23 至 33 行会在数据区中创建跳转表，形如第 19 至 21 行所示。第 36 行输出“.text”，表示接下来的内容为代码区。由于已在第 7 行把 SRC1 加载到寄存器 reg 中，我们就可在汇编指令中使用该寄存器的名称，为此我们要在第 37 行把中间指令里的 SRC1 改为相应的寄存器。第 38 行调用 ClearRegs()对 x86 CPU 中的寄存器进行了必要的回写，第 41 行调用 PutASMCode()产生了形如“jmp \*swtchTable1(%eax,4)”的汇编指令。

```

1 static void EmitIndirectJump(IRInst inst) {
2     //<IJMP, [BB4, BB5, BB6, NULL], t0, NULL>
3     BBlock *p; Symbol swtch;
4     int len; Symbol reg;
5     SRC1->ref--;
6     p = (BBlock *)DST;
7     reg = PutInReg(SRC1);
8     PutString("\n");
9     // .data
10    Segment(DATA);
11    CALLOC(swtch);
12    swtch->kind = SK_Variable;
13    swtch->ty = T(POINTER);
14    swtch->name = FormatName(
15        "swtchTable%d", SwitchTableNum++);
16    swtch->sclass = TK_STATIC;
17    swtch->level = 0;
18    /*****
19    swtchTable1:    .long    .BB4
20                    .long    .BB5
21                    .long    .BB6
22    *****/
23    DefineGlobal(swtch);
24    DST = swtch;
25    len = strlen(DST->aname);
26    //PRINT_DEBUG_INFO(( "%s ", DST->aname));
27    while (*p != NULL) {
28        DefineAddress(( *p )->sym);
29        PutString("\n");
30        LeftAlign(ASMFile, len);
31        PutString("\t");
32        p++;
33    }
34    PutString("\n");
35    // .text
36    Segment(CODE);
37    SRC1 = reg;
38    ClearRegs();
39    // jmp *swtchTable1(%eax,4)
40    // TEMPLATE(X86_IJMP,      "jmp *%0(%1,4)")
41    PutASMCode(X86_IJMP, inst->opds);

```

```
42 }
```

图 6.15 EmitIndirectJump()

### 6.3.4 为函数调用与返回产生汇编代码

在这一小节中，我们来讨论一下如何为函数调用和函数返回生成汇编代码。函数调用对应的中间指令如下所示：

```
//中间指令的四元式: < opcode, DST, SRC1, SRC2>
<CALL, 用于接收返回值的变量 recv, 函数名 func, 参数列表[arg1,arg2, ...,argn]>
```

根据 C 函数的调用约定，我们需要把参数从右向左入栈（即从 argn 到 arg1 依次入栈），不妨记这些参数所占用的栈内存为 stksize 字节。当函数调用返回后，主调函数要负责把这些参数出栈，这可通过形如“addl stksize,%esp”的汇编指令来实现。主调函数在产生 call 汇编指令之前，要对 eax、ecx 和 edx 这 3 个寄存器进行必要的回写操作。而 ebx、esi 和 edi 这 3 个寄存器则由被调函数负责保存，UCC 编译器会在所有函数的入口处保存这几个寄存器。为了加快返回值的传递，我们会尽量把返回值放在寄存器中。在 x86 平台上，按 C 标准的约定：

- (1) 若返回值为整型，则存于 eax 寄存器中(我们只考虑 32 位平台);
- (2) 若返回值为浮点型，则存于 x87 栈顶寄存器中;
- (3) 按 C 的语法规则，返回值不可以是数组类型。如果返回值是 1、2 或 4 字节的结构体对象，则存于寄存器 eax 中。若是 8 字节的结构体对象，则存于[edx: eax]中。若返回值是其他大小的结构体对象，则 C 编译器会为函数添加结构体指针作为第 1 个参数，如下所示。

```
typedef struct Data{
    int num[8];           //共 32 字节
} dt;
dt = GetData();
//经 C 编译器处理后，真正执行的函数调用为:
GetData(&dt);
```

因此，我们可按以下步骤来翻译形如<CALL, recv, func, [arg1,arg2, ...,argn]>的中间指令，其对应的 C 函数调用相当于是“recv= func(arg1, ..., argn)”。

- (1) 参数从右至左，即从 argn 到 arg1 依次入栈。
- (2) 对{eax,ecx,edx}这几个寄存器进行必要的回写操作。
- (3) 当返回值是结构体对象，且大小不是{1, 2, 4, 8}时，我们要取该结构体对象的地址并入栈。

(4) 若 func 为函数名 myadd，则产生形如“call myadd”的汇编指令；如果 func 是函数指针 fptr，则产生形如“call \* fptr”的汇编指令。

(5) 根据入栈参数所占的内存总和，调整寄存器 esp，即生成形如“addl stksize, esp”的汇编指令，其中 stksize 代表所有参数总共占用的栈内存大小。

(6) 根据返回值的类型从相应寄存器中取出返回值。对于大小不是{1, 2, 4, 8}的结构体对象，不需要由主调函数来取返回值。此时 UCC 已把形如“dt = GetData()”的函数调用改为“GetData(&dt)”，被调函数会从栈中取出&dt，从而实现结构体对象 dt 的赋值。

我们举个例子来说明上述过程。例如，对以下 C 程序中的函数调用 myadd()而言，UCC 编译器生成的汇编代码如下所示：

```
int myadd(int a,int b);
result = myadd(num1,num2);
/////////////////对应汇编代码/////////////
pushl num2          //参数 num2 入栈
pushl num1          //参数 num1 入栈
```

```

call myadd          //函数调用
addl $8, %esp      //所有参数出栈
movl %eax, result //取返回值

```

在此基础上，我们来看一下用于生成这些汇编代码的函数 EmitCall()，如图 6.16 所示。第 7 至 12 行把参数从右到左依次入栈，第 13 至 15 行调用 SpillReg() 函数对寄存器 eax、ecx 和 edx 进行必要的回写。当返回值是大小不落在 {1, 2, 4, 8} 中的结构体对象时，我们会在第 19 行取“返回值接收对象 recv”的地址，然后在第 20 行将该地址入栈。第 23 行用于产生函数调用指令，形如“call myadd”或者“call \* fptr”，第 25 至 28 行会把所有参数出栈。

```

1 static void EmitCall(IRInst inst) {
2   // <opcode, DST, SRC1, SRC2>
3   // <CALL, recv, func, [arg1, ..., argn]>
4   Vector args;    ILArg arg;  Type rty;
5   int i, stksize; args = (Vector)SRC2;
6   stksize = 0;    rty = inst->ty;
7   for (i = LEN(args) - 1; i >= 0; --i) {
8     arg = GET_ITEM(args, i);
9     PushArgument(arg->sym, arg->ty);
10    stksize += ALIGN(arg->ty->size,
11                      STACK_ALIGN_SIZE);
12  }
13  SpillReg(X86Regs[EAX]);
14  SpillReg(X86Regs[ECX]);
15  SpillReg(X86Regs[EDX]);
16  if (IsRecordType(rty)) //大小不是{1,2,4,8}的结构体对象地址入栈
17    && IsNormalRecord(rty)){
18    Symbol opds[2]; opds[0] = GetReg();
19    opds[1] = DST; PutASMCode(X86_ADDR, opds);
20    PutASMCode(X86_PUSH, opds);
21    stksize += 4;    DST = NULL;
22  } //产生"call myadd"或 "call * fptr"指令
23  PutASMCode(SRC1->kind == SK_Function
24            ? X86_CALL : X86_ICALL, inst->opds);
25  if(stksize != 0){ //addl $8, %esp
26    Symbol p;  p = IntConstant(stksize);
27    PutASMCode(X86_REDUCEF, &p);
28  }
29  if(DST == NULL){ //不需要浮点数返回值时,
30    if (IsRealType(rty)){ //也要弹出 x87 栈顶寄存器
31      PutASMCode(X86_X87_POP, inst->opds);
32    }
33    return;
34  }
35  if (IsRealType(rty)){ //从 x87 取浮点数返回值
36    PutASMCode(X86_STF4+(rty->categ != FLOAT),
37                inst->opds);
38    return;
39  }
40  switch (rty->size){ //取整型返回值或者大小为{1,2,4,8}的结构体对象
41  case 1: Move(X86_MOVI1, DST, X86ByteRegs[EAX]);
42    break;
43  case 2: Move(X86_MOVI2, DST, X86WordRegs[EAX]);
44    break;

```

```

45 case 4: AllocateReg(inst, 0);
46     if (DST->reg != X86Regs[EAX]) {
47         Move(X86_MOV14, DST, X86Regs[EAX]);
48     }
49     ModifyVar(DST);      break;
50 case 8: Move(X86_MOV14, DST, X86Regs[EAX]);
51     Move(X86_MOV14, CreateOffset(T(INT),
52                                     DST, 4, DST->pcoord), X86Regs[EDX]);
53     break;
54 default: assert(0);
55 }
56 }

```

图 6.16 EmitCall()

当返回值为浮点数时，如果主调函数不需要该返回值，我们要在第 31 行把 x87 栈顶寄存器弹出，以避免 x87 的寄存器栈过满。如果主调函数需要浮点数返回值，则通过第 35 至 39 行从 x87 栈顶寄存器中取出返回值，并弹出 x87 栈顶寄存器。而第 40 至 55 行则用于从寄存器 eax 或 edx 中获取“整数返回值”或者“大小为{1, 2, 4, 8}的结构体返回值”。

接下来，我们来分析一下图 6.16 第 9 行调用的函数 PushArgument()，其代码如图 6.17 第 1 至 24 行所示。第 3 至 5 行压入 float 类型的参数，而第 6 至 8 行压入 double 类型的参数，第 9 至 20 行用于把结构体对象复制到栈中，第 17 行的 opds[1] 记录要复制的字节数 ty->size，第 18 行的 opds[2] 是在栈中为结构体对象预留的内存大小，opds[2] 要大于或等于 opds[1] 的大小。而当参数是占 4 字节空间的整数时，我们通过第 22 行将其入栈。

```

1 static void PushArgument(Symbol p, Type ty) {
2     int tcode = TypeCode(ty);
3     if (tcode == F4) {
4         //pushl %%ecx;flds %0;fstps (%esp)
5         PutASMCode(X86_PUSHF4, &p);
6     }else if (tcode == F8) {
7         //subl $8, %%esp;fldl %0;fstpl (%esp)
8         PutASMCode(X86_PUSHF8, &p);
9     }else if (tcode == B) {
10        Symbol opds[2]; SpillReg(X86Regs[ESI]);
11        SpillReg(X86Regs[EDI]);
12        SpillReg(X86Regs[ECX]);
13        // leal %0, %%esi;subl %2, %%esp;
14        // movl %%esp, %%edi;
15        // movl %1, %%ecx;rep movsb
16        opds[0] = p;
17        opds[1] = IntConstant(ty->size);
18        opds[2] = IntConstant(ALIGN(ty->size,
19                                     STACK_ALIGN_SIZE));
20        PutASMCode(X86_PUSHB, opds);
21    }else{// pushl %0
22        PutASMCode(X86_PUSH, &p);
23    }
24 }
25 ****
26 *ptr = number;
27 <IMOV,ptr,number,NULL>
28 //////////汇编代码///////////
29 movl ptr, %eax

```

```

30  movl num, %ecx
31  movl %ecx, (%eax)
32 ****
33 static void EmitIndirectMove(IRInst inst) {
34     Symbol reg;
35     reg = PutInReg(DST);
36     inst->opcode = MOV;
37     DST = reg->next;
38     // <MOV, (%eax), number, NULL>
39     EmitMove(inst);
40 }
41 ****
42 t2 :*ptr;
43 <DEREF, t2, ptr, NULL>
44 //////////////汇编代码/////////////
45 movl ptr, %eax
46 movl (%eax), %ecx
47 ****
48 static void EmitDeref(IRInst inst) {
49     Symbol reg;
50
51     reg = PutInReg(SRC1);
52     inst->opcode = MOV;
53     SRC1 = reg->next;
54     assert(SRC1->kind == SK_IRegister);
55     EmitMove(inst);
56     ModifyVar(DST);
57     return;
58 }

```

图 6.17 PushArgument()

在图 6.17 第 26 行给出了形如 “\*ptr = number;” 的中间指令，UCC 称这样的指令为 IndirectMove，我们会在第 35 行把 ptr 加载到寄存器中，不妨设其为 eax，然后在 36 至 37 行把形如<IMOV,ptr,number,NULL>的中间指令改为<MOV,(%eax),number,NULL>，再通过第 19 行的 EmitMove()函数，就可以为 “\*ptr = number;” 产生以下汇编代码。我们已在前面的章节中分析过 EmitMove()函数，这里不再重复。

```

movl num, %ecx
movl %ecx, (%eax)

```

与此类似的，图 6.17 第 48 至 58 行的 EmitDeref()函数用来处理形如 “t2: \*ptr” 的中间指令，对应的四元式为<DEREF, t2, ptr, NULL>。我们先在第 51 行把 ptr 加载到寄存器中，不妨设其为 eax，第 52 至 53 行会把中间指令<DEREF, t2, ptr, NULL>改为<MOV, t2, (%eax), NULL>，之后通过第 55 行的 EmitMove()函数产生以下汇编代码：

```
movl (%eax), %ecx ; //临时变量 t2 对应的寄存器为 ecx
```

当遇到 C 程序里的形如 “return expr;” 的函数返回语句时，UCC 编译器会产生以下中间代码。其中，expr 代表 C 程序员编写的表达式，而 retVal 是 UCC 编译器生成的用于存放返回值的临时变量。当然，经过 UCC 编译器的中间代码优化后，retVal 也可能是 C 程序员命名的变量，而非临时变量。

```

return expr;
////////////对应中间代码////////////
<RET, retVal, NULL, NULL> //中间指令 RET 只是准备好返回值
<JMP, exitBB, NULL, NULL> //跳往函数的唯一出口

```

在函数的唯一出口对应的基本块 exitBB 中，UCC 编译器会通过 EmitEpilogue() 函数产生以下汇编代码，用于从被调函数返回到主调函数。

```
exitBB:
    movl %ebp, %esp
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
```

因此，中间指令<RET, retVal, NULL,NULL>所要完成的工作只是传递返回值，相关代码如图 6.18 所示。可分为以下几种情况来讨论：

(1) 返回值为浮点数。此时，按照约定，我们要把返回值保存在 x87 栈顶寄存器中。当然，如果返回值还不在 x87 栈顶寄存器内，就通过第 9 行调用 PutASMCode() 函数把返回值加载到 x87 栈顶寄存器中。

(2) 返回值是结构体对象，但大小不是{1, 2, 4, 8}。在这种情况下，在函数调用时，UCC 编译器就已经把形如“dt = GetData(a,b)”的函数调用改写为“GetData(&dt,a,b)”。此时隐式添加的第一个参数就是用于存放返回值的内存空间的首地址，通过第 33 行可取出该参数。而通过第 31 至 33 行，我们会把形如<RET, dt2, NULL, NULL>的中间指令改为<IMOV,&dt, dt2, NULL>，之后我们可在第 34 行调用 EmitIndirectMove() 函数进行结构体对象之间的复制操作。第 16 至 28 行的注释对此做了说明。

(3) 返回值是整数或者返回值是大小落在{1, 2, 4, 8}中的结构体对象。此时，可通过图 6.18 第 37 至 55 行用于把返回值传送到“寄存器 eax 或者 edx”中。

```
1 static void EmitReturn(IRInst inst) {
2     // <RET, retVal, NULL,NULL>
3     Type ty = inst->ty;
4     if (IsRealType(ty)){//返回值为浮点数
5         if (X87Top != DST){
6             int tcode = TypeCode(ty);
7             SaveX87Top();
8             // TEMPLATE(X86_LDF4, "flds %0")
9             PutASMCode(X86_LDF4 + tcode - F4,
10                     inst->opds);
11        }
12        X87Top = NULL;
13        return;
14    }
15    *****
16    1. 在函数调用时:
17        dt = GetData(a,b);          //由 C 程序员编写
18        被 C 编译器隐式地改为
19        GetData(&dt,a,b);
20        .....
21    2. 在函数返回时:
22        return dt2;                //由 C 程序员编写
23        /////经 UCC 优化后的中间代码/////
24        <RET, dt2, NULL, NULL>
25        <JMP, exitBB, NULL,NULL>
26        ////在汇编代码生成时，被 UCC 可改为以下中间代码/////
27        <IMOV, &dt, dt2,NULL>
28        <JMP,exitBB,NULL,NULL>
```

```

29  ****
30  if (IsRecordType(ty) && IsNormalRecord(ty)){
31      inst->opcode = IMOV;
32      SRC1 = DST;
33      DST = FSYM->params;
34      EmitIndirectMove(inst);
35      return;
36  }
37  switch (ty->size){
38  case 1: // "movb %1, %0"
39      Move(X86_MOV11, X86ByteRegs[EAX], DST);
40      break;
41  case 2: // "movw %1, %0"
42      Move(X86_MOV12, X86WordRegs[EAX], DST);
43      break;
44  case 4: // "movl %1, %0"
45      if (DST->reg != X86Regs[EAX]){
46          Move(X86_MOV14, X86Regs[EAX], DST);
47      }
48      break;
49  case 8:
50      Move(X86_MOV14, X86Regs[EAX], DST);
51      Move(X86_MOV14, X86Regs[EDX],
52          CreateOffset(T(INT), DST, 4, DST->pcoord));
53      break;
54  default:
55      assert(0);
56  }
57 }

```

图 6.18 EmitReturn()

### 6.3.5 为类型转换产生汇编代码

在这一小节中，我们来讨论一下整型和浮点型之间的类型转换。有些类型转换并不需要在汇编层次进行数据转换，例如 int 和 unsigned int 之间的转换只是改变了表达式的类型，对数据本身并无影响，以下表达式“(unsigned int) a”对应的二进制数据为 0xFFFFFFFF，而表达式“a”对应的二进制数据也为 0xFFFFFFFF。但对相同内容的二进制数据来说，进行“有符号整数的右移”和“无符号整数的右移”的结果是不一样的。换言之，对于 int 和 unsigned int 之间的转换，我们只要在语义检查时保存转型后的新类型即可，由于其数据本身并没有发生变化，我们不需要产生任何进行数据格式转换的汇编代码。

```

int a = -1;
unsigned int b;
b = ((unsigned int) a)>> 30;    //b 为 3
b = a >> 30;                  // b 为 0xFFFFFFFF

```

还有一点需要注意的是，UCC 编译器总是把低于 int 的整型先隐式地提升为 int，然后再进行其他转换。如下所示，当要进行“short --> char”的转换时，我们会以 int 作为中转，实际进行的转换为“short --> int --> char”。

```

short s = 3;
char c=(char)s;
////////////////////////////对应的中间代码///////////////////
t0 : (int)(short)s;      // short 先提升为 int

```

```

c = (char)(int)t0;      //再把 int 截断为 char
////////////////////////////对应的汇编代码///////////////////
movswl s, %eax        //两字节的 short 扩展为 4 字节的 int
movb %al, c            //只取低字节赋给 c

```

因此，在 UCC 编译器的类型转换中，“需要通过汇编代码来进行数据格式转换的”主要有以下几种，其中 In 分别代表占 n 字节的有符号整数，Un 分别代表占 n 字节的无符号整数，F4 代表 float，F8 代表 double。

(1) “I1、U1、I2 或者 U2” 提升为 I4。由“单字节或双字节”的整数进行符号位扩展，得到 4 字节的整数。C 程序员可以进行 “U1 -->U2”的转换，但 UCC 在语义检查阶段已改为 “U1 --> int --> U2”的转换。在汇编代码生成时不会遇到 “U1 --> U2 ”。

(2) “I4 或者 U4” 截断为 “I1 或者 U1”。只取 “4 字节的整数” 低 8 位，得到一个单字节的数据。

(3) “I4 或者 U4” 截断为 “I2 或者 U2”。只取 “4 字节的整数” 低 16 位，得到一个双字节的数据。

(4) “I4 或者 U4” 转换为 “F4 或者 F8”。通过 x87 协处理器进行数据格式转换，将整型转换为浮点数。

(5) “F4 或者 F8” 转换为 I4 或者 U4。通过 x87 协处理器进行数据格式转换，将浮点数转换为整数。

(6) “F4” 与 “F8” 之间互相转换。通过 x87 协处理器进行数据格式转换，实现 float 与 double 之间的转换。

当然，在一些没有浮点协处理器的中低端嵌入式平台上，浮点数运算需要由 CPU 通过整数运算来得到，这一般会由相应的浮点运算函数库来实现。下面，我们通过一个简单的例子来熟悉一下用于类型转换的汇编代码，如图 6.19 所示。第 15 行的 movsbl 指令用于对 i1 进行符号位扩展，从而得到 4 字节的 int；第 18 行的 movb 指令用于传送一个字节的数据，第 20 行的 movw 用于传送两个字节的数据。

```

1  char i1; unsigned char u1;
2  short i2; unsigned short u2;
3  int i4;   unsigned int u4;
4  float f4;  double f8;
5  int main(int argc,char * argv[]){
6    i4 = i1;
7    i1 = i4;
8    i2 = i4;
9    f4 = i4;
10   i4 = f4;
11   f4 = f8;
12   return 0;
13 }
14 //////////////////汇编代码/////////////////
15  movsbl i1, %eax // i4 = i1;
16  movl %eax, i4
17  movl i4, %eax // i1 = i4;
18  movb %al, i1
19  movl i4, %eax // i2 = i4;
20  movw %ax, i2
21  pushl i4          // f4 = i4;
22  fldl (%esp)
23  fstps f4
24  addl $4, %esp
25  flds f4          // i4 = f4;
26  subl $16, %esp
27  fnstcw (%esp)
28  movzwl (%esp), %eax
29  orl $0x0c00, %eax
30  movl %eax, 4(%esp)
31  fldcw 4(%esp)
32  fistpl 8(%esp)
33  fldcw (%esp)
34  movl 8(%esp), %eax
35  addl $16, %esp
36  movl %eax, i4
37  fldl f8          // f4 = f8
38  fstps f4

```

图 6.19 类型转换对应的汇编代码

图 6.19 第 21 至 24 行用于实现 int 到 float 的数据转换，我们在第 21 行把 i4 的值入栈，第 22 行把栈内存中的整数加载到 x87 中，第 23 行把经 x87 转换后的浮点数赋值给 f4，第 24 行进行 esp 寄存器的调整以保持栈的平衡，这几行汇编代码是由以下模板得来，并非最

优，但简单易懂。

```
TEMPLATE(X86_CVTI4F4, "pushl %1;fldl (%esp);fstps %0;addl $4, %%esp")
```

图 6.19 第 37 行从内存加载 double 型浮点数到 x87 协处理器中，第 38 行把转换后的 float 数据存到 f4 中。而第 25 至 36 行的汇编代码用于实现 float 到 int 的数据转换，其中很大一部分代码用来设置 x87 的控制寄存器，从而设定“四舍五入”的精度。我们已在第 1 章的图 1.36 中分析过这些汇编代码，这里不再重复。第 25 至 36 行的汇编代码是根据以下模板产生的，其中用到了寄存器 eax，因此在产生这些汇编代码前，我们需要先对寄存器 eax 进行必要的回写操作。

```
TEMPLATE(X86_CVTF4I4,
"flds %1;subl $16, %%esp;fnstcw (%esp);movzwl (%esp), %%eax;"  
"orl $0x0c00, %%eax;movl %%eax, 4(%esp);fldcw 4(%esp);fistpl 8(%esp);"  
"fldcw (%esp);movl 8(%esp), %%eax;addl $16, %%esp")
```

接下来，我们来讨论一下用于生成这些汇编代码的函数 EmitCast()，如图 6.20 所示。第 9 至 20 行用于实现 {I1, U1, I2, U2} 到 I4 的整型提升，对应的中间指令形如 <EXTI1, DST, SRC1, NULL>。我们会在第 6 行把目的操作数 DST 保存到局部变量 dst 中，第 12 行调用 AllocateReg() 函数为 DST 分配一个 4 字节寄存器，如果 DST 不是临时变量，则分配不成功。此时，我们需要在第 14 行直接调用 GetReg() 函数来得到一个寄存器，并重新设置中间指令里的操作数 DST，这会导致 “DST != dst” 。不论如何，在第 16 行，我们已得到一个寄存器，不妨记为 Rx，第 16 行通过形如 movsbl 的汇编指令进行符号位扩展，并把结果存到 Rx 中。如果“原先的目的操作数 dst”不是临时变量，则我们还要在第 18 行把寄存器 Rx 中的值传送给 dst，最终可得到形如 “movsbl i1, %eax; movl %eax, i4” 这样的汇编代码。

```
1 #define DST inst->opds[0]
2 #define SRC1 inst->opds[1]
3 #define SRC2 inst->opds[2]
4 static void EmitCast(IRInst inst) {
5     Symbol dst, reg;    int code;
6     dst = DST;  reg = NULL;
7     code = inst->opcode + X86_EXTI1 - EXTI1;
8     switch (code) {
9         case X86_EXTI1: case X86_EXTI2:
10        case X86_EXTU1: case X86_EXTU2:
11            // I1,U1,I2,U2 --> I4, 整型提升
12            AllocateReg(inst, 0);
13            if (DST->reg == NULL) {
14                DST = GetReg();
15                TEMPLATE(X86_EXTI1,"movsbl %1,%0");
16                PutASMCode(code, inst->opds);
17                if (dst != DST) {
18                    Move(X86_MOVI4, dst, DST);
19                }
20                break;
21        case X86_TRUI1:// 截断 I4/U4 --> I1
22            if (SRC1->reg != NULL) {
23                reg = X86ByteRegs[SRC1->reg->val.i[0]];
24            }
25            if (reg == NULL) {
26                reg = GetByteReg();
27                Move(X86_MOVI4,
28                     X86Regs[reg->val.i[0]], SRC1);
29            }
```

```

30     Move(X86_MOVI1, DST, reg); break;
31 case X86_TRUI2://截断 I4/U4 --> I2
32     .....      break;
33 case X86_CVTI4F4:  case X86_CVTI4F8:
34 case X86_CVTU4F4:  case X86_CVTU4F8:
35     //I4/U4 --> F4/F8, 整型转为浮点型
36 //TEMPLATE(X86_CVTI4F4,
37 //"pushl %1;fildl (%esp);fstps %0;addl $4,%esp")
38     PutASMCode(code, inst->opds);
39     break;
40 default:
41     SaveX87Top();
42     if(code != X86_CVTF4 && code != X86_CVTF8){
43         // F4/F8 --> I4/U4,浮点型转为整型
44         SpillReg(X86Regs[EAX]);
45         AllocateReg(inst, 0);
46         PutASMCode(code, inst->opds);
47         if ((DST->reg && DST->reg != X86Regs[EAX])
48             || DST->reg == NULL){
49             Move(X86_MOVI4, DST, X86Regs[EAX]);
50         }
51     }else{// double 与 float 之间互转
52         PutASMCode(code, inst->opds);
53     }
54     X87Top = NULL;      break;
55 }
56 ModifyVar(dst);
57 }

```

图 6.20 EmitCast()

图 6.20 第 21 至 30 行用于实现 {I4, U4} 到 I1 的截断操作，我们面对的中间指令形如 <TRUI1, DST, SRC1, NULL>。如果 SRC1 的值已经在寄存器中，不妨设其为 eax，则我们可在第 23 行得到 eax 的低 8 位对应的寄存器 al；否则，我们在第 26 行获取一个单字节寄存器，不妨设其为 al，第 27 行会把“占 4 字节的操作数 SRC1 的值”传送到寄存器 eax 中，由于 eax 寄存器的低 8 位就是寄存器 al，因此我们可在第 30 行把 al 的值传送给目的操作数 DST，最终可产生形如“movl i4, %eax; movb %al, i1;”的汇编代码。而 {I4, U4} 到 I2 的截断操作与此类似，我们在第 32 行省略了相关代码。

图 6.20 第 33 至 39 行用于实现整型到浮点型的转换，我们在第 38 行调用 PutASMCode() 函数，根据相应的模板来产生汇编代码即可。第 42 至 50 行用于实现浮点型到整型的转换，我们面对的中间指令形如 <CVTF4I4, DST, SRC1, NULL>，其中 DST 为整数，而 SRC1 为浮点数。由于在 X86\_CVTF4I4 模板中用到了寄存器 eax，因此我们要在第 44 行调用 SpillReg() 函数对 eax 进行必要的回写操作，第 45 行为 DST 分配一个 4 字节寄存器，不妨记为 Rx，第 46 行根据 X86\_CVTF4I4 等指令模板产生相应的汇编代码，实现浮点数到整数的转换，结果保存在寄存器 eax 中。如果中间指令里的操作数 DST 不是临时变量，或者 DST 是临时变量但所分配的寄存器不是 eax，则我们要在第 49 行把 eax 寄存器中的值传送给 DST。

### 6.3.6 为取地址产生汇编指令

在这一小节中，我们来讨论一下以下两条中间指令的翻译：

(1) 取地址指令 <ADDR, DST, SRC1, NULL>

例如 <ADDR, t0, number, NULL>, 表示取 number 的地址并保存到临时变量 t0 中

(2)对象清零指令<CLR, DST, SRC1, NULL>

例如<CLR, arr, 16, NULL>, 表示把 arr 所占 16 字节的内存清零

我们先举一个例子来说明, 对于图 6.21 第 4 行局部数组 arr 的初始化来说, 我们可以先通过第 11 行的 CLR 中间指令把数组 arr 所占 16 字节内存清零, 然后再通过第 12 行的 MOV 指令对 arr[0]进行赋值。第 21 至 26 行是 CLR 指令对应的汇编代码, 我们在汇编中调用库函数 memset()实现了对象清零的操作。而对于第 5 行的 “ptr1 = &num1;” 而言, 我们可以通过第 13 行的 ADDR 中间指令来取 num1 的地址, 并保存于临时变量 t0 中, 再通过第 14 行的 MOV 指令对 ptr1 进行赋值。第 28 行 “leal num1, %eax” 是 ADDR 指令对应的汇编代码。

1 int num1,num2;	19
2 int * ptr1, * ptr2;	20 //////////////汇编代码/////////////
3 void f(void){	21 pushl \$16 // arr:16
4 int arr[4] = {10};	22 pushl \$0
5 ptr1 = &num1;	23 leal -16(%ebp), %eax
6 ptr2 = &num2;	24 pushl %eax
7 *ptr1 = *ptr2;	25 call memset
8 }	26 addl \$12, %esp
9 //////////////////中间代码////////////////	27 movl \$10,-16(%ebp) //arr[0]=10;
10 function f	28 leal num1, %eax // t0:&num1;
11 arr : 16; // CLR 指令	29 movl %eax, ptr1 // ptr1 = t0
12 arr[0] = 10;// MOV 指令	30 leal num2, %ecx // t1:&num2;
13 t0 :&num1; // ADDR 指令	31 movl %ecx, ptr2 // ptr2 = t1;
14 ptr1 = t0; // MOV 指令	32 movl ptr2, %edx // t3: *ptr2
15 t1 :&num2; // ADDR 指令	33 movl (%edx), %ebx
16 ptr2 = t1; // MOV 指令	34 movl ptr1, %edx // *ptr1 = t3
17 t3 :*ptr2; // DEREF 指令	35 movl %ebx, (%edx)
18 *ptr1 = t3; // IMOV 指令	

图 6.21 取地址和对象清零的例子

与取地址指令有关的中间指令还有第 17 行的 DEREF 指令和第 18 行的 IMOV 指令, 我们已在前面的章节中介绍过这两条中间指令的翻译。通过这两条中间指令, 我们可以实现第 7 行的 C 语句 “\*ptr1 = \*ptr2;” 所需的语义。

接下来, 我们来看看用于产生相应汇编代码的函数 EmitAddress()和函数 EmitClear(), 如图 6.22 所示。第 5 行调用 AllocateReg()函数为 ADDR 指令里的临时变量 DST 分配一个寄存器, 第 7 行产生汇编指令把 SRC1 的地址加载到 DST 对应的寄存器中, 第 8 行调用 ModifyVar()来设置临时变量 DST 的回写标志位。

```

1 // <ADDR, t0, number, NULL>
2 static void EmitAddress(IRInst inst) {
3     assert(DST->kind == SK_Temp
4             && SRC1->kind != SK_Temp);
5     AllocateReg(inst, 0);
6     //TEMPLATE(X86_ADDR,"leal %1, %0")
7     PutASMCode(X86_ADDR, inst->opds);
8     ModifyVar(DST);
9 }
10 // <CLR, arr, 16, NULL>
11 static void EmitClear(IRInst inst){
12     int size = SRC1->val.i[0];
13     Symbol p = IntConstant(0);
14     switch (size){
15         case 1:

```

```

16     //TEMPLATE(X86_MOV11,"movb %1,%0")
17     Move(X86_MOV11, DST, p);
18     break;
19 case 2:
20     //TEMPLATE(X86_MOV12,"movw %1, %0")
21     Move(X86_MOV12, DST, p);
22     break;
23 case 4:
24     //TEMPLATE(X86_MOV14,"movl %1, %0")
25     Move(X86_MOV14, DST, p);
26     break;
27 default:
28     SpillReg(X86Regs[EAX]);
29     //TEMPLATE(X86_CLEAR,
30     // "pushl %1;pushl $0;leal %0, %%eax;""
31     // "%eax;call memset;addl $12,%%esp")
32     PutASMCode(X86_CLEAR, inst->opds);
33     break;
34 }
35 }

```

图 6.22 EmitAddress() 和 EmitClear()

图 6.22 第 10 至 35 行用于为 CLR 指令产生汇编代码，当要清零的对象大小为{1, 2, 4}字节时，我们可以在第 15 至 22 行产生 mov 汇编指令来实现清零，否则我们就通过第 32 行产生汇编代码，在其中调用 memset() 函数来实现对象清零。由于在汇编指令模板 X86\_CLEAR 中使用了寄存器 eax，我们需要在第 28 行调用 SpillReg() 函数对寄存器 eax 进行回写。

至此，我们完成了为各中间指令产生汇编代码的工作。最后，让我们再看一下 Compile() 函数，如图 6.23 所示，这是我们从第 3 章开始一路走来的路线图，我们历经了“语法分析”、“语义检查”、“中间代码生成及优化”和“汇编代码生成”这几个阶段。

```

1 static void Compile(char *file){
2     AstTranslationUnit transUnit;
3     //初始化
4     Initialize();
5     //语法分析
6     transUnit = ParseTranslationUnit(file);
7     if (ErrorCount != 0)
8         goto exit;
9     //语义检查
10    CheckTranslationUnit(transUnit);
11    if (ErrorCount != 0)
12        goto exit;
13    if (DumpAST) {
14        //输出语法树
15        DumpTranslationUnit(transUnit);
16    }
17    //中间代码生成与优化
18    Translate(transUnit);
19    if (DumpIR) {
20        //输出中间代码
21        DAssemTranslationUnit(transUnit);
22    }
23    // 汇编代码生成
24    EmitTranslationUnit(transUnit);

```

```

25 exit:
26 //收尾
27 Finalize();
28 }

```

图 6.23 Compile()

## 6.4 本章习题

1. 用 UCC 编译运行以下程序，结合图 6.16 和图 6.18，分析由 UCC 编译器生成的语法树、中间代码和汇编代码。

```

#include <stdio.h>
struct Data{
    int num[16];
}dt;
struct Data GetData(void){
    struct Data dt2;
    return dt2;
}
int main(int argc,char * argv[]){
    dt = GetData();
    return 0;
}

```

2. 用 UCC 编译运行以下程序，对实验结果进行分析。请结合 UCC 编译器生成的汇编代码，画出栈内存示意图，分析以下程序存在什么样的问题。

```

#include <stdio.h>
int * getPtr(void){
    int number = 2015;
    return &number;
}
int main(int argc,char * argv[]){
    int * ptr = getPtr();
    int number;
    number = *ptr;
    printf("number = %d \n",number);
    number = *ptr;
    printf("number = %d \n",number);
    return 0;
}

```

3. 在中间代码生成阶段，UCC 编译器为基本块取的名字形如“BB1”，但在汇编代码生成时，出现在汇编程序中的基本块为什么是“.BB1”。

## 参考文献

- [1] Wenjun Wang. UCC[EB/OL], <http://sourceforge.net/projects/ucc/>.
- [2] Technical Committee X3J11. Ansi.c.txt[EB/OL], <http://flash-gordon.me.uk/ansi.c.txt> .
- [3] Alfred V. Aho, Monica S. Lam, Ravi Sethi,et al. Compilers: Principles, Techniques, and Tools [M]. New Jersey: Pearson Education, Inc, 2006.
- [4] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual[EB/OL],<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html> .
- [5] Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language[M]. New Jersey: Prentice Hall, 1988.
- [6] 邓志. x86/x64 体系探索及编程[M]. 北京: 电子工业出版社, 2012.
- [7] 高博,范德成. Stanley B.Lippman 谈 C++语言和软件产业的发展[EB/OL]  
<http://www.csdn.net/article/2013-11-27/2817632>.
- [8] 陈意云,张昱. 编译原理[M]. 北京: 高等教育出版社, 2008.
- [9] Stanley B. Lippman 著. 侯捷译. 深度探索 C++对象模型 [M]. 武汉: 华中科技大学出版社, 2007.

# 图的清单

图 1.1 分析树.....	2
图 1.2 为非终结符 T 编写语法分析函数.....	3
图 1.3 a+a+a 对应的分析树.....	3
图 1.4 Program 文法.....	4
图 1.5 GetToken().....	7
图 1.6 PrimaryExpression().....	8
图 1.7 MultiplicativeExpression().....	10
图 1.8 左结合和右结合的语法树.....	10
图 1.9 AdditiveExpression().....	11
图 1.10 表达式对应的分析树和语法树.....	11
图 1.11 VisitArithmeticNode().....	12
图 1.12 复杂声明的分析树与语法树.....	14
图 1.13 输出类型信息.....	15
图 1.14 二维数组对应的分析树与语法树.....	16
图 1.15 PostfixDeclarator().....	17
图 1.16 ExpressionStatement().....	18
图 1.17 IfStatement().....	20
图 1.18 While 和复合语句.....	21
图 1.19 VisitStatementNode().....	22
图 1.20 main().....	22
图 1.21 编译工具链.....	25
图 1.22 编译命令.....	27
图 1.23 Execute().....	28
图 1.24 用于说明数据区的 C 程序.....	31
图 1.25 静态数据区及函数 h()对应的汇编代码.....	31
图 1.26 栈示意图.....	33
图 1.27 main()函数对应的汇编代码.....	34
图 1.28 arith.c.....	35
图 1.29 初始化和未初始化的变量.....	35
图 1.30 arith.s.....	36
图 1.31 有条件跳转指令.....	37
图 1.32 整数比较.....	38
图 1.33 有符号和无符号整数.....	39
图 1.34 浮点数运算.....	39
图 1.35 x87 浮点芯片的数据寄存器栈.....	40
图 1.36 浮点数运算的汇编代码.....	40
图 1.37 由 x87 实现浮点数到整数的转换.....	41
图 1.38 NaN 和 Inf.....	42
图 1.39 x87 状态寄存器与浮点数比较的结果.....	43
图 1.40 浮点数比较.....	43

---

图 1.41 数组、指针和结构体.....	44
图 1.42 函数调用.....	45
图 1.43 printf()函数.....	48
图 1.44 内存布局示意图.....	49
图 1.45 OurPrintfV1().....	50
图 1.46 OurPrintfV2().....	50
图 1.47 Do_Error().....	51
图 1.48 Do_Error 的栈示意图.....	52
图 2.1 Makefile.....	54
图 2.2 make all 和 make test.....	55
图 2.3 UCL main().....	55
图 2.4 Compile().....	56
图 2.5 ReadSourceFile().....	57
图 2.6 struct input.....	58
图 2.7 GetNextToken().....	59
图 2.8 SetupLexer().....	59
图 2.9 ScanIdentifier().....	60
图 2.10 InternName().....	61
图 2.11 CurrentHeap.....	63
图 2.12 struct heap 结构体.....	64
图 2.13 struct mblock 链表.....	64
图 2.14 HeapAllocate().....	65
图 2.15 struct type.....	66
图 2.16 类型结构.....	67
图 2.17 结构体的类型描述.....	67
图 2.18 结构体的类型结构.....	68
图 2.19 struct Data 对象 dt 的内存布局.....	68
图 2.20 函数的类型描述.....	69
图 2.21 旧式风格和新式风格的函数.....	70
图 2.22 函数的类型结构.....	70
图 2.23 SetupTypeSystem().....	71
图 2.24 被误当作旧式风格的函数.....	72
图 2.25 栈示意图.....	73
图 2.26 struct symbol.....	74
图 2.27 各种不同类别的符号.....	75
图 2.28 variableSymbol.....	76
图 2.29 公共子表达式.....	76
图 2.30 valueDef 和 valueUse.....	77
图 2.31 TrackValueChange().....	77
图 2.32 struct table.....	78
图 2.33 valueNumTable 哈希表.....	78
图 2.34 TryAddValue().....	79
图 2.35 AddSymbol().....	80

图 2.36 C 语言的作用域.....	80
图 2.37 多个作用域的符号表.....	81
 图 3.1 struct astNode.....	84
图 3.2 struct astExpression.....	84
图 3.3 ParseExpression().....	85
图 3.4 表达式 “a,b,c” 对应的语法树.....	85
图 3.5 ParseAssignmentExpression().....	86
图 3.6 “a=b=c” 对应的语法树.....	87
图 3.7 ParseConditionalExpression().....	87
图 3.8 “a?b:c” 对应的语法树.....	88
图 3.9 二元运算符表达式.....	88
图 3.10 ParseBinaryExpression()版本 1.....	90
图 3.11 ParseBinaryExpression()版本 2.....	91
图 3.12 两种不同版本的 ParseBinaryExpression 对应的分析树.....	91
图 3.13 ParseUnaryExpression().....	92
图 3.14 &a 对应的语法树.....	93
图 3.15 ParseUnaryExpression()_sizeof.....	93
图 3.16 IsTypename().....	95
图 3.17 sizeof(a)对应的语法树.....	95
图 3.18 ParseUnaryExpression()_cast.....	96
图 3.19 (int) a 对应的语法树.....	96
图 3.20 ParsePostfixExpression().....	97
图 3.21 后缀运算符对应的语法树.....	97
图 3.22 ParsePrimaryExpression().....	98
图 3.23 (a+2015+b)对应的语法树.....	99
图 3.24 ParseStatement().....	100
图 3.25 ParseLabelStatement().....	101
图 3.26 标号语句对应的语法树.....	101
图 3.27 if 语句和 switch 语句的语法分析.....	102
图 3.28 ParseCompoundStatement().....	103
图 3.29 首符集.....	104
图 3.30 复合语句对应的语法树.....	104
图 3.31 PostCheckTypedef().....	105
图 3.32 CheckTypedefName().....	106
图 3.33 ParseTranslationUnit().....	107
图 3.34 TranslationUnit 对应的语法树.....	108
图 3.35 ParseExternalDeclaration().....	109
图 3.36 声明 Declaration 所对应的语法树.....	110
图 3.37 GetFunctionDeclarator().....	111
图 3.38 FunctionDefinition.....	112
图 3.39 函数定义所对应的语法树.....	113
图 3.40 PreCheckTypedef().....	114
图 3.41 ParseDeclaration().....	115

---

图 3.42 ParseCommonHeader()	118
图 3.43 ParseCommonHeader() 构造的语法树	119
图 3.44 声明说明符的产生式	120
图 3.45 ParseDeclarationSpecifiers()	121
图 3.46 由 ParseDeclarationSpecifiers() 构造的语法树	121
图 3.47 ParseStructOrUnionSpecifier()_产生式	122
图 3.48 ParseStructOrUnionSpecifier()_左大括号	123
图 3.49 ParseStructOrUnionSpecifier() 构建的语法树	123
图 3.50 ParseStructDeclaration()	124
图 3.51 ParseStructDeclaration() 所构建的语法树	125
图 3.52 ParseStructDeclarator()	125
图 3.53 声明符 Declarator 对应的产生式	126
图 3.54 ParseDirectDeclarator()	128
图 3.55 ParsePostfixDeclarator()	129
图 3.56 ParseParameterTypeList()	130
图 3.57 ParseInitDeclarator()	130
图 3.58 ParseInitializer()	131
图 3.59 {{10,20,30},40,50} 对应的语法树	131
图 4.1 语义检查的例子	133
图 4.2 CheckExpression()	135
图 4.3 CheckBinaryExpression()	135
图 4.4 Adjust()	136
图 4.5 数组索引的例子	138
图 4.6 ScalePointerOffset()	139
图 4.7 数组寻址	140
图 4.8 CheckPostfixExpression()	141
图 4.9 字符串	143
图 4.10 CheckPrimaryExpression()	144
图 4.11 AddString()	145
图 4.12 语法树的变化示意图	145
图 4.13 CheckFunctionCall()	147
图 4.14 函数调用的语法树	148
图 4.15 CheckArgument()	149
图 4.16 CanAssign()	150
图 4.17 Cast()	152
图 4.18 转型所对应的语法树	153
图 4.19 函数体中的函数声明	153
图 4.20 LookupId() 函数	155
图 4.21 与符号有关的数据结构	155
图 4.22 AddFunction() 和 AddVariable()	157
图 4.23 CheckMemberAccess()	158
图 4.24 成员选择运算的语法树	159
图 4.25 TransformIncrement()	159

---

图 4.26 IsCompatibleType()	161
图 4.27 IsCompatibleFunction()	163
图 4.28 CompositeType()	164
图 4.29 CommonRealType()	165
图 4.30 CheckUnaryExpression()	167
图 4.31 CheckTypeCast()	168
图 4.32 提领运算	168
图 4.33 CheckUnaryExpression()_OP_DEREF	170
图 4.34 CheckUnaryExpression_case_OP_ADDR	171
图 4.35 宏 PERFORM_ARITH_CONVERSION	172
图 4.36 CheckEqualityOP()	174
图 4.37 CheckAddOP() 和 CheckSubOP()	175
图 4.38 CheckAssignmentExpression()	177
图 4.39 a+=b 的语法树	177
图 4.40 CheckConditionalExpression()	178
图 4.41 CheckStatement()	179
图 4.42 CheckCompoundStatement()	181
图 4.43 CheckSwitchStatement()	182
图 4.44 CheckTranslationUnit()	184
图 4.45 CheckDeclarationSpecifiers()	186
图 4.46 CheckDeclarator()	187
图 4.47 CheckFunctionDeclarator()	189
图 4.48 CheckParameterTypeList()	191
图 4.49 IsIncompleteType()	192
图 4.50 类型信息链表	193
图 4.51 DeriveType()	194
图 4.52 int *arr1[5] 和 int(*arr2)[5] 的类型结构	194
图 4.53 为结构体创建类型结构	195
图 4.54 CheckStructOrUnionSpecifier()	196
图 4.55 StartRecord()	198
图 4.56 CheckStructDeclaration()	199
图 4.57 AddOffset()	200
图 4.58 EndRecord()	201
图 4.59 struct initData	203
图 4.60 BORBitField()	204
图 4.61 CheckInitializer()	205
图 4.62 CheckInitializerInternal()	207
图 4.63 CheckInitializerInternal()_ARRAY	208
图 4.64 CheckInitializerInternal()_STRUCT	209
图 4.65 CheckInitConstant()	211
图 4.66 Linkage	213
图 4.67 CheckGlobalDeclaration()	215
图 4.68 CheckFunction()	216

图 5.1 基本块.....	218
图 5.2 与中间代码有关的数据结构.....	221
图 5.3 Translate().....	223
图 5.4 基本块的静态结构和动态结构.....	223
图 5.5 布尔表达式和算术表达式.....	225
图 5.6 TranslateBranchExpression().....	226
图 5.7 生成跳转指令.....	228
图 5.8 较复杂的短路运算.....	228
图 5.9 TranslateBranch().....	230
图 5.10 GenerateMove().....	234
图 5.11 PeepHole().....	235
图 5.12 取地址与偏移.....	237
图 5.13 对数组元素的寻址.....	238
图 5.14 Offset().....	241
图 5.15 数组元素和结构体成员的翻译.....	244
图 5.16 TranslateFunctionCall().....	245
图 5.17 赋值表达式的语法树.....	246
图 5.18 TranslateAssignmentExpression().....	248
图 5.19 a++和++a 的语法树.....	249
图 5.20 TranslateIncrement().....	250
图 5.21 TranslateUnaryExpression().....	251
图 5.22 TranslateConditionalExpression().....	252
图 5.23 TranslateIfStatement().....	253
图 5.24 TranslateCompoundStatement().....	255
图 5.25 switch 语句的例子.....	257
图 5.26 Case 语句密度与二分查找.....	258
图 5.27 TranslateSwitchStatement().....	260
图 5.28 TranslateSwitchBuckets().....	262
图 5.29 循环语句的翻译方案.....	263
图 5.30 删除无用的临时变量.....	264
图 5.31 Optimize().....	266
图 5.32 ExamineJump().....	267
图 5.33 无代码的空基本块.....	268
图 5.34 基本块合并的 5 种情况.....	269
图 5.35 TryMergeBBlock().....	270
图 6.1 汇编代码生成的例子.....	272
图 6.2 EmitTranslationUnit().....	274
图 6.3 LayoutFrame().....	276
图 6.4 GetAccessName().....	277
图 6.5 寄存器分配的例子.....	279
图 6.6 GetRegInternal().....	280
图 6.7 EmitBBlock().....	282
图 6.8 由中间指令到汇编代码的过程.....	284

---

图 6.9 为中间指令里的 I4 或 U4 类型操作数分配寄存器.....	286
图 6.10 EmitMove().....	288
图 6.11 EmitX87Move 和 EmitMoveBlock.....	290
图 6.12 EmitAssign().....	292
图 6.13 EmitX87Assign().....	293
图 6.14 EmitBranch()和 EmitJump().....	295
图 6.15 EmitIndirectJump().....	297
图 6.16 EmitCall().....	299
图 6.17 PushArgument().....	300
图 6.18 EmitReturn().....	302
图 6.19 类型转换对应的汇编代码.....	303
图 6.20 EmitCast().....	305
图 6.21 取地址和对象清零的例子.....	306
图 6.22 EmitAddress()和 EmitClear().....	307
图 6.23 Compile().....	308

# 表的清单

表 1.1 常用的条件跳转指令..... 37

表 3.1 与声明有关的非终结符..... 118

# 尾声

总有曲终人散时，不知不觉我们已经完成了对 UCC 编译器的剖析，一路走来，最深的体会仍然是“纸上得来终觉浅，绝知此事要躬行”。按这个道理，理解 UCC 编译器的最好办法应是“直接阅读其源代码，思考 UCC 编译器在不同的执行点应处于怎样的状态，加入一些打印语句，输出相应的调试信息来验证自己的判断是否正确，如果发现 Bug，就写一些测试程序来触发 Bug，然后修改 UCC 编译器的源代码”。在遇到困惑时，或许拙作能带来一点点的帮助和提示，但拙作不能代替，也不应取代对 UCC 源代码的主动阅读。源代码分析的相关书籍很容易带来一个副作用，即读者在书籍的牵引下被动地看书，而忽视了对源代码本身的主动阅读。主动思考和被动阅读所带来的效果是完全不一样的。

LLVM 和 GCC 的功能已经足够强大，也有足够庞大的程序员社区在支撑其不断发展和改进。而 UCC 几乎没有可能与这些编译器媲美，UCC 编译器最大的优点在于“它的规模和复杂度都在一个 C 程序员单枪匹马就可以掌控的范围内”。只要我们愿意付出努力，我们一定可以较好地搞明白 UCC 编译器，从而更好地用好 C 语言这个简洁有力的利器。UCC 编译器最适合的读者可能是在校的大学生，有时间，有精力，且暂时还不用考虑“写代码养家糊口”，可以静下心来打好专业基本功。能有机会在一线参与编译器开发的程序员少之又少，但攻克 UCC 编译器这座山头的意义在于“它能让我们获得攻克其他山头的能力”，这可能就是所谓磨刀不误砍柴功。当然，选择怎样的山头来练兵，源于我们自己的兴趣。从打基本功的角度出发，我们可以选择自己感兴趣的其他山头来攻克，比如麻省理工大学的 XV6 操作系统，而不一定要选择 UCC 编译器。XV6 操作系统是 UnixV6 系统在 X86 平台上的重生。

每个人心中都有自己的偶像，当我们在谈论某某是我的偶像时，其潜台词往往是“我们希望自己能成为那样的人”。绝大部分人都无法达到自己偶像的高度，但偶像很大程度上会指引我们前进的方向。20 世纪 70 年代诞生的 C 和 Unix，影响了一代又一代的程序员，对 IT 整个行业也产生了深远的影响。阅读“UCC 源代码和 XV6 源代码”，比“捧着编译原理和操作系统的教材”，应会更有趣得多，这也是我们向伟大的先行者 Dennis Ritchie 和 Ken Thompson 致敬的最好办法，其实也是学习编译原理和操作系统的最有效方法之一。有了源代码级的感性认识后，再去补充理论知识，才会得到更好的升华。

CPU 由于物理上的限制，其主频已不再无限提高了，摩尔定律在单 CPU 上已渐渐失效，现在和未来的方向应是多核、分布式和并行。在分布式或者并行的环境下，如何为程序员提供一个易用的编程环境，或许是未来若干年编译方向的研究热点。当然还有优化，毕竟我们始终在追求“没有最好，只有更好”。

限于水平和能力，书中不当之处敬请批评与指正。